ZAMA

THRESHOLD (FULLY) HOMOMORPHIC ENCRYPTION

Carl Bootland[3] (iD)   Kelong Cong[3] (iD)   Daniel Demmler[3] (iD)
Tore Kasper Frederiksen[3] (iD)   Benoît Libert[3]
Jean-Baptiste Orfila[3] (iD)   Dragos Rotaru[2] (iD)   Nigel P. Smart[1,3] (iD)
Titouan Tanguy[3] (iD)   Samuel Tap[3] (iD)   Michael Walter[3] (iD)

[1] COSIC, KU Leuven, Leuven, Belgium
[2] Gateway, New York, USA
[3] Zama, Paris, France

Version 0.1
April 17, 2025

1 Abstract

This document is a **preliminary** version of what is intended to be submitted to NIST by Zama as part of their threshold call. The document also serves as **partial** documentation of the protocols used in the Zama MPC system for threshold TFHE.

However, note that the Zama software includes many optimizations built on top of the simple specifications given here. In particular the TFHE parameters given here are larger than those used by the Zama software. This is because the Zama TFHE library contains optimizations which are beyond the scope of this document. Thus the parameters given in this document are compatible with the description of TFHE given here, and take no account of the extra optimizations in the Zama software.

Also note that we describe more protocols than that provided in the Zama software. In particular this document describes BGV and BFV threshold implementations, MPC–in–the–Head based proofs of correct encryption.

We present mechanisms to perform *robust* threshold key generation and decryption for Fully Homomorphic Encryption schemes such as BGV, BFV and TFHE, in the case of super honest majority, $t < n/3$, or $t < n/4$, in the presence of malicious adversaries.

The main mechanism for threshold decryptions follow the noise flooding principle, which we argue is sufficient for BGV and BFV. For TFHE a more subtle technique is needed to apply noise flooding, since TFHE parameters are small. To deal with all three FHE scheme, and obtain a unified framework for all such schemes, we are led to consider secret sharing over Galois Rings and not just finite fields.

We consider two sets of threshold profiles, depending on whether $\binom{n}{t}$ is big or small. In the small case we obtain for all schemes an *asynchronous* protocol for *robust* threshold decryption, and we obtain a *robust synchronous* protocol for threshold key generation; both with $t < n/3$. For the large case we only support TFHE, and our protocols require an "offline phase" which requires *synchronous* networks and can "only" tolerate $t < n/4$.

The threshold key generation operation, and the above mentioned offline phase, require access to a generic offline MPC functionality over arbitrary Galois Rings. This functionality is fully specified here.

Finally, we present Zero–Knowledge proof techniques for proving the valid encryption of an FHE ciphertext. These proofs are important in a number of application contexts.

2 Executive Summary

In this document we discuss methodologies for **threshold decryption** and **threshold key generation** for the popular Fully Homomorphic Encryption (FHE) schemes of BGV [BGV12], BFV [FV12, Bra12] and TFHE [CGGI16, CGGI20, BdBB+25]; although our focus is mainly on threshold protocols suitable for TFHE. We provide **robust** protocols[1], i.e. they can recover from injected adversarial errors, which guarantee output delivery. Our protocols utilize an **asynchronous network** in the online phase of the protocols. For some threshold profiles (for when either the number of players is large or for threshold key generation) we require an offline phase; in such situations the offline phase needs to assume a **synchronous network**.

The protocols are built upon a (relatively) generic, robust **Multi–Party Computation** (MPC) function–ality which works over **Galois Rings**. For $\mathfrak{n}$ players and $\mathfrak{t}$ corruptions, this comes in two variants:

1. A variant, for when $\binom{\mathfrak{n}}{\mathfrak{t}}$ is small, which utilizes pseudo–random secret sharing. We support in this parameter region threshold decryption for BGV, BFV and TFHE. The threshold decryption works over asynchronous networks and requires $\mathfrak{t} < \mathfrak{n}/3$. For threshold key generation we require an offline phase, which also requires $\mathfrak{t} < \mathfrak{n}/3$, but which requires a synchronous network.

2. A variant, for when $\binom{\mathfrak{n}}{\mathfrak{t}}$ is large. We support in this threshold protocols only for TFHE; both threshold decryption and threshold key generation use an online phase which works over asynchronous networks and requires $\mathfrak{t} < \mathfrak{n}/3$. However, both also require an offline phase which requires $\mathfrak{t} < \mathfrak{n}/4$ and synchronous networks.

In addition we provide three forms of **zero–knowledge proofs** for the correct encryption of FHE ciphertexts in the above schemes. Such proofs are needed to ensure that any input ciphertexts to a protocol are not adversarially generated. This is a major issue when building MPC–like protocols utilizing FHE which may have adversarially generated input, see [Sma23] for a full discussion of such protocols. We provide three such forms of Zero–Knowledge proofs; two which are pre–quantum secure (for soundness but not zero–knowledge), have short proofs, but a relatively long prover time, and a third variant which is fully post–quantum secure, has longer proofs, but a shorter prover time. We note, a more efficient (in terms of proof size and prover time) post–quantum proof system may be possible using systems such as Ligero [AHIV17, AHIV23, BFH+20].

For the rest of this section we outline the document in these four directions: MPC over Galois Rings, threshold decryption, threshold key generation, and zero–knowledge proofs in more detail.

2.1 Generic MPC over Galois Rings

To enable the our threshold operations for FHE we require an MPC engine which works either modulo $q = 2^k$ or modulo $q = p_1 \ldots p_k$ (where p_i are largish primes). The former is for TFHE and the latter is for BGV and BFV. Most work on MPC considers MPC defined over a finite field, and in neither of the cases of interest to us is $\mathbb{Z}/(q)$ a finite field. This leads us to need to build an MPC engine over a Galois Ring.

This has been done before (theoretically) in the literature in various cases [ACD+19, JSvL22]. These latter works usually consider $q = 2^k$, generalizing them to the case of $q = p_1 \ldots p_k$ is relatively straight forward. We adapt these prior works, and apply them to the [DN07] robust MPC protocol. The [DN07] MPC protocol was originally defined for finite fields; we provide the relatively trivial generalization to arbitrary Galois Rings for our choices of q.

As described above, in doing so we introduce two classes of threshold profiles (see Table 1 later for a concise overview):

- For small values of $\binom{\mathfrak{n}}{\mathfrak{t}}$ (we call such profiles *nSmall*) we obtain (apart from a small setup procedure) a robust MPC protocol, which works over asynchronous networks, and requires $\mathfrak{t} < \mathfrak{n}/3$. This

---

[1]Sometimes in the literature what we refer to as robust protocols are called protocols with Guaranteed Output Delivery (GOD–protocols)

protocol is new, but a relatively simple adaption of existing techniques.

- For large values of $\binom{\mathfrak{n}}{\mathfrak{t}}$ (called profiles *nLarge*) we obtain, using a direct application of the method of [DN07], a robust MPC protocol which works over asynchronous networks in the online phase, and requires $\mathfrak{t} < \mathfrak{n}/3$. However, the offline phase works only over synchronous networks and requires $\mathfrak{t} < \mathfrak{n}/4$.

One could produce in the *nLarge* regime a protocol which works for $\mathfrak{t} < \mathfrak{n}/3$ in the offline phase, by fully adapting the methodology in [DN07]. However, the dispute resolution system for such a protocol is relatively complex, and so we decided not to pursue this extension.

2.2   Threshold FHE Decryption

The problem of threshold decryption for FHE schemes, henceforth called threshold–FHE, is as old as FHE itself. The problem is for a set of $\mathfrak{n}$ parties to have a secret sharing of the underlying FHE secret key so that they can between them decrypt a given FHE ciphertext correctly, in the case where at most $\mathfrak{t}$ of the parties are corrupt. Indeed, Gentry's original thesis [Gen09] mentioned threshold–FHE as a way of utilizing FHE to perform a very low round complexity semi–honest MPC protocol.

To understand the technical problem with threshold–FHE it is worth considering the "format" of a simple FHE scheme. To explain we utilize the format of BFV/TFHE [FV12, CGGI16, CGGI20, BdBB$^+$25] ciphertexts, but a similar discussion can be provided for other FHE schemes such as BGV. Consider encrypting an element $m \in \mathbb{Z}/(P)$, using a standard Learning–With–Errors (LWE) ciphertext of the form $(\mathbf{a}, b)$ with ciphertext modulus q, where $\mathbf{a} \in (\mathbb{Z}/(Q))^{\ell}$ and $b \in \mathbb{Z}/(Q)$, using the equation

$$b = \mathbf{a} \cdot \mathbf{s} + e + \Delta \cdot m \quad (\mathrm{mod}\ Q)$$

where $\Delta = \lfloor Q/P \rfloor$, e is some "noise" term and $\mathbf{s} \in (\mathbb{Z}/(Q))^{\ell}$ is the secret key. Usually, in the FHE setting, $\mathbf{s}$ is chosen to be a vector with small norm, for example $\mathbf{s} \in \{0, 1\}^{\ell}$.

To enable threshold–FHE we first secret share the secret key $\mathbf{s}$ among $\mathfrak{n}$ parties, a process which we shall denote by $\langle \mathbf{s} \rangle$ to signal a sharing modulo Q with respect to a threshold $\mathfrak{t} < \mathfrak{n}$ linear secret sharing scheme. On input of the ciphertext $(\mathbf{a}, b)$ we can then produce trivially a secret sharing of the value $e + \Delta \cdot m$ by computing

$$\langle t \rangle = b - \mathbf{a} \cdot \langle \mathbf{s} \rangle = \langle e + \Delta \cdot m \rangle.$$

By opening the value of $\langle t \rangle$, all parties can then perform rounding to obtain m. However, this reveals the value of e, which combined with the ciphertext and the message, will reveal information about the secret key $\mathbf{s}$.

There are two ways around this problem. The first is to add some additional noise into the secret sharing before opening (a process called noise flooding), the second method is to extract the message m from the shared value $\langle t \rangle$ using a generic MPC protocol.

2.2.1   Threshold Decryption via Noise Flooding

In this method the decrypting parties somehow generate an additional secret shared noise term $\langle E \rangle$, and the value which is opened is now

$$\langle t \rangle = b - \mathbf{a} \cdot \langle \mathbf{s} \rangle + \langle E \rangle = \langle e + E + \Delta \cdot m \rangle.$$

We require that E should introduce enough randomness to mask the e value after the shared value $\langle t \rangle$ is opened. If E is too small then too much information about e is revealed, if E is too big then the final rounding will not reveal the correct value of m. Diagrammatically we can consider this process as approximated by the diagram in Figure 1.

4
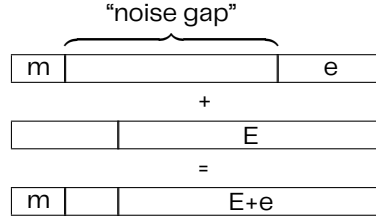
Figure 1: Representation of the noise addition for threshold decryption.

To mask, statistically, all information in e we would (naively) require E to be chosen uniformly from a range which is 2^{dist} larger than e. Thus if we can bound the ciphertext noise by $|e| < B$, then we would require E to be chosen uniformly in the range $[-2^{dist} \cdot B, \ldots, 2^{dist} \cdot B]$. This process is often dubbed "noise flooding" in the literature. However, this would mean we require $\Delta > 2^{dist} \cdot B$, which in turn seems to imply that the ciphertext modulus Q needs to be "large". The work effort of the adversary to distinguish the distributions in the underlying security proof is then 2^{dist}. Prudently this would lead us to select $dist = 80$.

In [DDE$^+$23] it is shown that by adding an E term on, which is itself the addition of at least two uniform distributions in the range $[-2^{stat} \cdot B, \ldots, 2^{stat} \cdot B]$, the resulting work effort of the adversary to distinguish the distributions becomes $2^{2 \cdot stat}$. We can, hence, obtain enough security by selecting $stat \approx 40$, and so reduce the need for very much larger Q values.

In theory it may be possible to select E from a smaller range, and rely on some computational assumptions in order to argue about security. This approach is taken in two recent papers, [BS23] and [CSS$^+$22], via the Renyi divergence. This methodology enables parameters to be chosen in which Q is much smaller than the above analysis would require. However, this leads to additional problems in the larger protocols in which we embed our threshold decryption, see Section 4.10.1 for a full discussion on this. Thus the use of Renyi divergence is not without problems in this situation.

We contend that what seems like an increase in the size of Q is not a problem in practice.

- When using BFV/BGV in an SHE leveled mode the problem does not occur. In such schemes each level essentially adds an extra 20–40 bits (depending on the implementation) into the noise gap. Thus by simply decreasing the number of usable levels by a small constant (say, one or two) one can obtain a noise gap which is enough to apply the flooding technique. Thus in such schemes our methodology can be applied, without any need for prior processing of a ciphertext.

- When using BFV/BGV in a mode which enables the use of bootstrapping (which we do not consider in this document) we also do not need to increase the size of Q. Bootstrapping enables us to reduce the size of the noise e in the ciphertext $(\mathbf{a}, b)$ to be as small as possible. Thus if bootstrapping is performed, and the FHE scheme is such that the noise gap between e and m in Figure 1 is large enough, then the noise flooding methodology will work "out-of-the-box".

- Thus the only place where noise flooding is in practice a problem is when the FHE parameters are such that the noise gap is tiny, even after a bootstrapping operation is performed. This is exactly the situation in TFHE, where one (usually) selects a relatively small Q value (for example $Q = 2^{64}$). This small Q value, and associated small LWE dimension ℓ, requires the size of the noise even after bootstrapping to be around 2^{30} in order to ensure security. This means the noise gap is too small, but only by tens of bits. In such a situation we apply, following the method of [DDE$^+$23] a switch of the parameters (Q, ℓ) to slightly larger ones $(\overline{Q}, \overline{\ell})$ where $\overline{Q} = 2^{128}$. Combined with a bootstrapping operation, which is relatively efficient using TFHE, one obtains a noise gap which is suitable for use with the noise flooding approach.

With these methods we obtain a fully robust, one-round, threshold decryption protocol which works over asynchronous networks for our threshold profile *nSmall*, using no offline phase. For our threshold profiles *nLarge* the online phase of this threshold decryption protocol is also one round-and asynchronous, but

we requires a multi-round offline phase which requires us to assume synchronous networks.

2.2.2 Threshold Decryption via Generic–MPC

The above outlined approaches for BGV and BFV are perfectly acceptable in almost all situations. For TFHE the necessity to apply a bootstrap for the parameters $(\overline{Q}, \overline{\ell})$ can add an unwanted latency to the entire threshold decryption process. For Local Area Network (LAN) type applications, with short ping–times, one can trade computation for communication. In particular one can apply generic "bit–extraction" techniques from standard MPC–protocols in order to extract the message m from the sharing $\langle t \rangle$, without needing to apply to noise flooding.

This means we can keep all operations using the smaller modulus $Q = 2^{64}$, instead of $\overline{Q} = 2^{128}$. However, this method requires multiple rounds of communication between the parties. Since each round costs us a latency of (at least) the parties ping–times, such a method may not be suitable in all situations.

The method of robust threshold decryption via generic–MPC for TFHE can be applied in both our two sets of threshold profiles *nSmall* and *nLarge*. With the final threshold decryption protocol inheriting the properties of the underlying MPC protocol (namely the required threshold and the assumptions with respect to network synchronicity).

2.3 Threshold Key Generation

Our methodology for threshold key generation is to utilize the generic MPC functionality described above. This is to enable the resulting FHE keys to have as close a noise distribution as possible to the non–threshold versions. One could apply a form of Multi–Party or Multi–Key FHE (MP–FHE/MK–FHE) style protocol to obtain a threshold key generation, but these are (usually) not robustly secure (which is our primary security target) and in addition they result in slightly different noise distributions. Such MP–FHE schemes are also usually for full threshold access structures (and hence cannot produce schemes which have robust threshold decryption). Having the noise distribution close to the optimal of the standard non–threshold scheme is also important for schemes such as TFHE, where small parameters are chosen and the parameters are highly optimized to obtain efficient bootstrapping operations.

To produce threshold FHE key generation within an MPC engine the main issue is how does one sample the small noise distributions for the underlying LWE problems. These usually come from discrete Gaussian distributions. Sampling from such distributions is inefficient within an MPC engine. Thus we replace the standard discrete Gaussian distributions with either the NewHope distribution [PAA+19] (in the case of BGV and BFV), or small uniform distributions in the case of TFHE. The use of the NewHope distribution is relatively standard for BGV and BFV. For TFHE the error distributions require far larger standard deviations (due to the smaller parameters), thus a uniform error distribution is easier to deal with. This makes very little difference to resulting security (as the lattice estimator [APS15] demonstrates), and also makes little difference in the final parameters. The number of secret shared random bits can be quite high, so in an Appendix we sketch a method which uses less secret shared random bits; but which pushes the FHE parameters slightly higher.

As noticed for key generation for BGV SPDZ keys in [RST+22], the key generation method is essentially just linear operations after one has determined secret sharings of the small LWE noise samples and the associated secret keys. This observation applies also to BFV and TFHE. Thus threshold key generation is obtained by a relatively simple MPC program for all cases, once enough random secret shared bits have been generated.

2.4 Zero–Knowledge Proofs

When deploying multi–party FHE solutions (i.e. protocols which utilize threshold decryption) a key security requirement is to provide zero–knowledge proofs of knowledge of correct encryptions for fresh

FHE ciphertexts. In this document we provide three forms of such zero-knowledge proofs. The first two are pre-quantum secure, they derive from work in [Lib24, BEL⁺24], and are based on pairings on elliptic-curve groups; they provide very short proofs but the time to generate a proof can be long. A quantum computer is able to break the soundness of these first proofs, but not the zero-knowledge property. Thus privacy of the inputs is maintained in the presence of a quantum computer, but a quantum adversary will be able to prove invalid statements. The third are fully post-quantum, and are based on MPC-in-the-Head techniques; they provide longer proof, but a shorter prover time. The MPC-in-the-Head proofs are based on the techniques from [FMRV22, FR23], which are themselves based on the technique from [KKW18]. Again, we note, more efficient post-quantum proof system may be possible using systems such as Ligero [AHIV17, AHIV23, BFH⁺20].

The first two (pre-quantum) proofs are more suitable for TFHE style FHE, as they require pairing friendly elliptic curves whose group order is larger than the ciphertext modulus, whereas the third (post-quantum) proofs can be applied to any form of FHE scheme. Our second pre-quantum proof is more efficient than the first, but it proofs a statement which has some soundness slack from the desired input statement. Such soundness slack needs to be factored into the parameters for the FHE scheme.

3 Index

List of Figures

List of Tables

List of Design Decisions

List of Parameter Choices

4 Clarification of Prior Work

In this section we enumerate all the building blocks we will be using from the existing literature. All of these building blocks are fully specified below in order to understand our full protocol stack.

4.1 Conventions

This document uses the following basic conventions and notations:

- For a set S, we denote by $a \leftarrow S$ the process of drawing a from S with a uniform distribution on the set S.
- If S contains a single element, i.e. $S = \{s\}$, then we sometimes use the shorthand $a \leftarrow s$, instead of $a \leftarrow S$.
- If D is a probability distribution, we denote by $a \leftarrow D$ the process of drawing a with the given probability distribution.
- For a probabilistic algorithm A, we denote by $a \leftarrow A$ the process of assigning a the output of algorithm A, with the underlying probability distribution being determined by the random coins of A.
- An array of bytes is indexed with zero being the least significant byte. Thus the integer 65534 consists of two bytes, $b_1 | b_0$ with b_1 being equal to the byte FE in hex, or 254 in decimal.
- A byte has eight bits $b_7 | b_6 | \ldots | b_1 | b_0$ with the least significant bit being b_0. Thus the byte value 17 has $b_0 = 1$ and $b_7 = 0$.
- The bit representation of an integer has the least significant bit in the right most position. Thus shifting the bit representation to the right is equivalent to dividing the integer by two.
- In converting an integer (u32, u128 etc) to a byte representation we assume a little–endian con-vention (compatible with x86 architectures).
- In converting a byte string to bits we interpret the bytes in left to right order, thus the array of bytes '3210' corresponds to the bit string '00000011|00000010|00000001|00000000'.
- If $\mathbf{a}$ is a vector of N elements then we write $\mathbf{a} = (a_1, \ldots, a_N)$, however we may also write $\mathbf{a}[i]$ to denote a_i. This is useful if we have a vector of vectors, i.e. we have ℓ vectors $\mathbf{a}_1, \ldots, \mathbf{a}_\ell$ and we write the j–th component of the i–th vector as $\mathbf{a}_i[j]$.
- We use the notation $\mathbb{Z}/(m) = \mathbb{Z}/m\mathbb{Z}$ to denote the ring of integers modulo m. The notation $\mathbb{Z}_p$ is reserved for the ring of p–adic integers, where p is a prime. This latter ring is the ring of formal power series in p, i.e.

$$\mathbb{Z}_p = \Big\{ \sum_{i=0}^{\infty} a_i \cdot p^i : a_i \in [0, \ldots, p-1] \Big\}$$

  where the set is equipped with the natural addition and multiplication operations.
- When using a ring such as $\mathbb{Z}/(m) = \mathbb{Z}/m\mathbb{Z}$ we will take a centred representation of the values, i.e.

$$\mathbb{Z}/(m) = \{ -\lceil m/2 \rceil + 1, \ldots, \lfloor m/2 \rfloor \}.$$

- When working with a ring such as $\mathbb{Z}/(p^k)$, one can view it as working with $\mathbb{Z}_p$ but restricting oneself to working with a p–adic precision of k digits. This view is convenient as we can simply take results for the p–adic ring, and then reduce them modulo $q = p^k$ to obtain an associated result for the ring $\mathbb{Z}/(q) = \mathbb{Z}/(p^k)$.
- To enable domain separation for functions inputs are often prefixed or suffixed with a byte string of a small number of characters. We shall denote this prefix/suffix by $DSep(str)$. So for example the byte string produced by $DSep(ABCD)$ is $0x41424344$.

4.2 Security Levels

We utilize three distinct security parameters in this document sec, $dist$ and $stat$. We outline their meaning here:

- sec is a traditional computational security parameter. It is used for primitives such as public–key encryption, block–ciphers and hash functions. A work effort of 2^{sec} is needed to effect a complete break of the underlying primitive.
- $dist$ is a traditional statistical security parameter. It measures distances between distributions, i.e. we may require two distributions D_1 and D_2 to lie within a statistical distance of 2^{-dist} of each other, or perhaps there is a random event which may leak information to the adversary which happens with probability $1/2^{dist}$, or the correctness of some algorithm is only ensured with probability $1 - 2^{-dist}$.
- $stat$ is a parameter which is used to mask a distribution in a way which requires the attacker to expend at least $2^{2 \cdot stat}$ effort in order to distinguish the masked distribution from a random one. In practice we would set $stat = dist/2$.

It is common these days to target a computational security level of $sec = 128$. Some parts of the academic Multi–Party Computation (MPC) literature target a statistical security level of $dist = 40$, however, we do not feel this is prudent and so prefer to set $dist = 80$. Which leaves us with setting $stat = dist/2 = 40$.

> **Parameter Choice 1: Security Parameters**
>
> We target values of $sec = 128$, $dist = 80$ and $stat = 40$ in this document. Higher values are possible if desired.

4.3 Actively Secure Robust MPC over Galois Rings

At the core of our protocols for threshold decryption and key generation, is a robust MPC protocol. An MPC protocol is parameterized by the number of players $\mathfrak{n}$ and the threshold $\mathfrak{t}$ of bad players which can be tolerated. Note, one can have MPC protocols which tolerate non–threshold adversary structures; for simplicity in this document we only consider threshold adversary structures.

4.3.1 General Introduction to MPC Protocols

MPC protocols have various design requirements; the basic ones being

1. **Correctness:** If all the parties follow the protocol then the parties obtain the correct evaluation of the function on their private inputs.
2. **Privacy:** The honest players private inputs are private (except what can be deduced from the output value) with respect to a given adversarial strategy (semi–honest vs active, static vs adaptive).
3. **Robustness:** If the dishonest parties are active (i.e. they are allowed to deviate from the protocol) then we require that the honest parties still receive the correct function output. Sometimes this property is called Guaranteed–Output–Delivery, or GOD.

These three properties, of correctness, privacy and robustness, can be obtained under three different security "levels".

1. **Perfect Security:** All three properties hold with probability one, irrespective of the power of the adversary.
2. **Statistical Security:** All three properties hold with probability exponentially close to one, irrespective of the power of the adversary.
3. **Computational Security:** The properties hold assuming some computational assumption on the adversary.

The first two of these levels are often combined and referred to as information theoretic security.

To complicate the design space even more we can divide adversaries into static ones (in which the adversary decides which parties to corrupt at the start of the protocol) versus adaptive ones (in which the adversary can decide which parties to corrupt as the protocol progresses). To provide simpler protocols, and avoid the need for things such as non-committing encryption in the private channels linking the parties in our MPC protocols we adopt an MPC model which assumes static corruptions.

> **Design Decision 1: Static vs Adaptive Adversaries**
>
> We assume static adversaries in our MPC security models.

However, in practice adversaries may not decide to play ball with such an assumption. Thus for our threshold FHE protocols we provide resharing algorithms, which allow a set of parties to either update the secret sharing amongst themselves, or transfer the secret sharing to a completely different set of parties (with a potentially different size, but still supporting $t < n/3$). Combined with mechanisms to perform secure erasures of existing secret shared data, this provides a form of pro-active security.

> **Design Decision 2: Proactive Security via Resharing**
>
> We provide protocols for resharing the underlying secret shared secret key for our threshold FHE protocols. This enables (when combined with a form of secure erasure) a pro-actively secure system to be deployed.

We can also make different assumptions on the underlying network; whether it is synchronous in nature versus whether it is asynchronous, and whether a broadcast channel is made available to the parties "for free". In practice broadcast channels do not come for free, one needs to built them out of point-to-point channels.

The classical MPC literature deals with MPC over finite fields. Our MPC protocols are based on secret sharing, of which the initial work, for finite fields, goes back to the BGW and CCD protocols in 1988, [BGW88] [CCD88]. The BGW and CCD protocols provide robust information theoretically secure protocols in the case of $t < n/3$. The BGW protocol is perfectly secure, whereas CCD is statistically secure. The bound of $t < n/3$ is theoretically the best possible for robust information theoretic protocols. If one is willing to assume computational assumptions then the theoretical bound of $t < n/3$ can be extended to $t < n/2$, whilst still obtaining robust security. A protocol realizing such a bound is the classic GMW protocol from 1987, [GMW87]. The bound of $t < n/2$ can also be obtained in the statistically secure scenario if one assumes a broadcast channel is provided to the parties, see [RB89]. Note, to concretely obtain such a broadcast channel in this scenario one, however, would need to assume $t < n/3$, due to classical bounds on Byzantine agreement.

The above works assume synchronous networks, if one wishes to examine asynchronous networks then the theoretical bounds become weaker. For example in [BCG93] it is shown one can obtain perfectly secure, robust, protocols when $t < n/4$ (and this is again a theoretical limit) in the asynchronous setting. If one is willing to relax to statistically secure robust protocols then one can obtain $t < n/3$ in the asynchronous setting, see [BKR94]. Relaxing to computational security also allows one to obtain $t < n/3$ as the best possible result in the asynchronous setting. This last result was demonstrated by [HNP05].

It is possible to weaken the active security guarantees from guaranteed output delivery, i.e. robust protocols, to a situation where, if the adversary deviates from the protocol, the honest parties abort. A variant of active-with-abort protocols are ones in which any aborting party is identified. This provides so-called active-with-identifiable-abort security. Protocols which are secure with respect to active-with-identifiable-abort are important if one wishes to stop active-with-abort protocols from suffering

from Denial-of-Service attacks. As our protocols are fully robust we automatically identify dishonest parties as the protocol progresses.

For active-with-abort protocols one can also obtain security for larger threshold, indeed all thresholds up to $t < n$, see for example the SPDZ family of protocols [DPSZ12, CDE$^+$18]. Actively secure protocols with identifiable-abort are often more complex to instantiate, so we do not consider them further here; after all our robust protocols automatically identify cheating parties in any case.

An even weaker form of security is to provide no security guarantees when a party deviates from the protocol, so called semi-honest security. One can obtain active-with-abort security for most MPC protocols at little additional marginal cost over semi-honest protocols. For example, in the regime of $t < n/2$, the protocol of [CGH$^+$18] provides active-with-abort security at a cost of running the underlying semi-honest protocol twice.

We aim for robustness, as we feel that in real applications active-with-abort it not enough (even if we have identifiable abort). If one does not have robustness of the underlying MPC protocol, then it needs to be left to a higher level application as to what to do when an adversary tries to deviate, or fails to respond, in a protocol. This can be difficult for the higher level application to deal with, or reason about, thus it makes sense to solve this problem at the underlying MPC protocol layer.

Design Decision 3: Level of Active Security

We aim for fully robust protocols, which guarantee output delivery, over protocols which are active-with-abort, or which have identifiable abort.

Our MPC protocols are going to be used to thresholdize a Fully Homomorphic Encryption (FHE) scheme, thus we are already going to make computational assumptions. However, MPC protocols which mainly use information theoretic constructs are often much faster than those requiring heavy computational assumptions. Thus we focus on MPC protocols which make limited use of computational assumptions; mainly lightweight cryptographic primitives such as hash functions and block-ciphers, and which have negligible error probabilities.

Design Decision 4: Computationally and Statistically Secure MPC

To achieve efficient protocols we are happy with protocols which have a negligible probability of error, and which utilize lightweight cryptographic assumptions.

The "modern" approach to secret sharing based MPC protocols is to divide the protocol into two phases; an offline phase (which is input independent) and an online phase (which does depend on the inputs). Such a division is made highly efficient using so-called Beaver Triples, which themselves are derived from the circuit-randomization technique of Beaver [Bea92]. The idea of splitting into phases can be dated back to other works of Beaver [Bea95, Bea97]; although the division into phases seems first to have been formally presented in [HN06] in the case of computationally secure protocols, and by [DN07] for information theoretically secure protocols. Practical instantiations of the offline/online paradigm date back to the start of practical MPC protocols; for example [DGKN09] present an offline/online protocol which is almost an asynchronous protocol (this seems to be the earliest record of a concrete implementation of a protocol using the offline/online paradigm).

Design Decision 5: Dividing MPC Protocols into Offline and Online Phases

Our MPC protocols are divided into a function independent offline phase and an online phase which executes the desired functionality.

As a final note on general MPC protocols; standalone MPC protocols for general application need

input sub-protocols to enable players to enter their data into the MPC system. In our application of MPC, players will have no inputs coming from outside the system, all values within the MPC system will be generated from within the MPC system. Thus we can simplify the design space by not needing to deal with input protocols in our underlying MPC system.

> **Design Decision 6: MPC with no Input**
>
> Our MPC protocols do not require any input sub-protocols to enter players private inputs into the system.

4.3.2 The Choice of Damgård–Nielsen as the Base MPC Protocol

Our core full MPC protocol is based on the protocol from [DN07], which is a robust protocol in the synchronous network setting. The basic strategy in designing our MPC protocol is to adopt the methodology from [DN07], which utilizes Damgård–Nielsen multiplication as opposed to Maurer multiplication to produce triples. With Damgård–Nielsen multiplication we do not need to protect against each player dealing an invalid Maurer sharing, this makes the robust triple generation protocol, in the case of $t < n/4$, very simple. Extending this bound to $t < n/3$ requires utilizing a more elaborate form of dispute control; which we decide to forego in general. We however provide a simple form of dispute control in our setting where $\binom{n}{t}$ is small.

The protocol of [BTH08], is perhaps conceptually simpler and avoids the need for probabilistic checks. But, the yield of triple production is much less in [BTH08] compared to [DN07]. Thus, the protocol in [DN07] is chosen as a basis, over the protocol of Beerliová–Trubíniová and Hirt [BTH08], as it offers very high yield in terms of triple production. We feel that in practice an improved efficiency of using [DN07] is more beneficial than a stronger (relatively theoretical) security guarantee of avoiding probabilistic correctness checks.

The work of Goyal et al, [GLS19], provides a more efficient (in terms of communication complexity) version of the methods of [BTH08, DN07]. However, this comes at the expense of some added complexity in terms of the overall construction on the "happy path" (when there are no errors introduced by the adversary). The simpler communication complexity only applies on the "unhappy path". Given "most" instances are likely to always stay on the happy path, as adversaries are actually rare (especially as we can identify who committed any adversarial behavior), we expect the happy path complexity to be the most important factor in determining system performance. Thus, we optimize for happy path complexity over dealing with the unhappy path. A similar low complexity on the unhappy path, robust protocol, in the case of $t < n/2$ is given in [GSZ20].

To provide even greater performance improvements we make minor tweaks to the [DN07] protocol by assuming the security of various lightweight cryptographic primitives, such as hash-functions, block-ciphers and so forth.

We also, to simplify exposition and also to reduce round complexity, use a non-batched/non-king version of the protocol in [DN07]. In practice this may be more efficient, at least for smallish values of n, than the batched version. This is because the batched version requires two rounds of interaction, whereas the non-batched version requires one round. The batched version also requires additional processing (to batch and unbatch the data). However, the batched version reduces the total amount of communication being sent by all parties. In Section 7.2.3.1 we give the modifications needed for the batched version, which should be asymptotically more efficient when n grows. The exact cross over point depends on the network latency and other factors.

> **Design Decision 7: No King Version of Protocols**
>
> We choose not to use the king paradigm for batched openings in our main exposition, this enables us to reduce round complexity, at the expense of increased communication complexity.

A problem with utilizing the protocol of [DN07] as a basis is that whilst the online phase is fully asynchronous when $\mathfrak{t} < \mathfrak{n}/3$, the offline phase requires synchronous networks. Finding a truly practically efficient asynchronous offline phase is a work of ongoing research, thus we settle (reluctantly) for a synchronous offline phase; for all protocols which require an offline phase (which is not all of them). One can (partially) justify such a synchronicity assumption for the offline phase, by assuming that the offline phase can be executed with a much larger packet timeout than the online phase; and thus messages from honest but slow parties can be assumed to be delivered in the offline phase.

> **Design Decision 8: Asynchronous vs Synchronous Communication**
>
> Our main MPC protocol is built using a synchronous offline phase and an asynchronous online phase.

4.3.3 Extending Damgård–Nielsen to Galois Rings

However, unlike [DN07], we require MPC protocols where the base arithmetic structure is not a finite field but a Galois Ring. In particular a ring such as $\mathbb{Z}/(q)$ where $q = 2^k$, or $q = p_1 \cdots p_k$. This change in the protocol if [DN07] is accomplished by changing the secret sharing scheme from one over a field, to one over a Galois Ring (a topic which we shall return to below).

The MPC protocol for finite fields from [DN07] can be easily mapped to the setting of secret sharing schemes over Galois Rings, using the same techniques as in [ACD+19]. This is done in much the same way as [ACD+19] used their techniques to map the related protocol of [BTH08] to the case of $q = 2^k$.

To utilize [DN07] we need to use many of the same tricks from [ACD+19], and in addition extend the probabilistic check from [DN07] to the ring setting (which we provide a security proof for in Theorem 5). This results in a Galois Ring based MPC protocol, which inherits the properties of the [DN07] protocol; namely that it is highly efficient on the so–called happy path (when no errors are detected).

4.3.4 Threshold Profiles *nSmall* and *nLarge*

We consider two families of threshold profiles, depending on the number of combinations of selecting $\mathfrak{t}$ items from $\mathfrak{n}$ things, i.e. $\binom{\mathfrak{n}}{\mathfrak{t}}$. One profile in which $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "small". Such profiles we say are in threshold profile *nSmall*. The other profiles are where $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "large"; for which we will call such threshold profiles *nLarge*. See Section 6.1 for a full discussion on these threshold profiles.

> **Design Decision 9: Threshold on Adversaries for MPC Protocols**
>
> We identify two different cases:
> - When $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "small" we target robust security with a threshold bound of $\mathfrak{t} < \mathfrak{n}/3$. We call such profiles *nSmall*.
> - When $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "large" we target robust security with a threshold bound of $\mathfrak{t} < \mathfrak{n}/4$. We call such profiles *nLarge*.
>
> The division between the *nSmall* and *nLarge* regimes will be defined by a value *nSmallBnd*, with $\binom{\mathfrak{n}}{\mathfrak{t}} < nSmallBnd$ denoting the regime for the *nSmall* threshold profiles.

The dispute resolution framework is particularly simple in threshold profile *nSmall*, so in this case we present a novel dispute control methodology which allows us to achieve $\mathfrak{t} < \mathfrak{n}/3$. In particular, we

describe a methodology to ensure that the multiplication triples produces are robust which is simpler than the more general approach when $\binom{\mathfrak{n}}{\mathfrak{t}}$ is large; this makes use of the fact that the production method inherent in the Pseudo–Random Secret Sharing (PRSS) construction is essentially self–authenticating (as $\mathfrak{t} < \mathfrak{n}/3$). This check, see Figure 57 for details, we cannot find in the prior literature, but it is obvious and trivial.

In threshold profile *nLarge*, we only target $\mathfrak{t} < \mathfrak{n}/4$, as the dispute control methodology when $\mathfrak{n}/4 \le \mathfrak{t} < \mathfrak{n}/3$ is relatively complex (see [DN07] for the outline of how this could be implemented). Our protocols could be extended to support $\mathfrak{t} < \mathfrak{n}/3$ in the *nLarge* threshold profile setting, but we feel the added complication is not worth it in practice (especially given our choice for *nSmallBnd* below).

There are other trade–offs between the *nSmall* and *nLarge* settings, apart from the whether we can support $\mathfrak{t} < \mathfrak{n}/3$ or $\mathfrak{t} < \mathfrak{n}/4$:

1. The *nSmall* settings requires a set–up phase of complexity $O(\binom{\mathfrak{n}}{\mathfrak{t}}) = O(nSmallBnd)$.
2. The *nLarge* settings require an expensive offline phase for all threshold FHE operations.
3. The *nSmall* setting enables a simple threshold decryption protocol, with no offline phase, for BGV, BFV and TFHE.
4. The *nSmall* setting allows the easy the generation of shared random bits when the MPC modulus is an odd composite number, and hence easy threshold key generation for BGV and BFV. In contrast for *nLarge* we do not provide a threshold key generation *or* decryption method for BGV and BFV.
5. The *nLarge* setting allows a simple offline phase to produce secret shared random bits, when the MPC modulus is a power of two. This enables the use of our the *nLarge* profile for our threshold key generation and decryption methodologies for TFHE.

In any specific instantiation these trade–offs could mean that the *nSmall* protocols are to be preferred over the *nLarge* protocols; or vice versa. However, in terms of this document the main distinction is that the choice of the cut–off, i.e. the value of *nSmallBnd*, will affect the parameter choices for the underlying FHE scheme. The slightly altered parameter choices enable us to support our threshold decryption procedures in the *nSmall* setting. Thus to fix parameters we need to select a maximum value of *nSmallBnd*.

> **Design Decision 10: Size of** *nSmallBnd*
>
> We fix our FHE threshold parameters so that one can tolerate *nSmallBnd* $= 10,000$, however an implementation may decide to choose a smaller value of this bound.

4.3.5 Alternative, discounted, MPC Choices

The main complexity in our offline phase comes from the need to obtain robustness. If we relaxed this then other possibilities are available. On the other hand we could also strengthen the requirements, opening up other protocol choices. In this section we briefly discuss some of these options, and why we decided to not go with them. To help summarize some of these choices we refer to Table 1. The value in blue is where our protocol sits.

Protocol	$\binom{\mathfrak{n}}{\mathfrak{t}}$	Offline Threshold	Offline Network	Offline Security	Online Threshold	Online Network	Online Security
[DN07]/This Work	*nSmall*	$\mathfrak{n}/3$	Synch	Robust	$\mathfrak{n}/3$	Async	Robust
[DN07]/This Work	*nLarge*	$\mathfrak{n}/4$	Synch	Robust	$\mathfrak{n}/3$	Async	Robust
[DN07]/This Work	*nSmall*	$\mathfrak{n}/3$	Synch	Robust	$\mathfrak{n}/2$	Sync	A–w–A
[DN07]/This Work	*nLarge*	$\mathfrak{n}/4$	Synch	Robust	$\mathfrak{n}/2$	Sync	A–w–A
Protocol 1 [JSvL22]	All	$\mathfrak{n}/2$	Synch	A–w–A	$\mathfrak{n}/3$	Async	Robust
[CGH+18]/Protocol 5 [JSvL22]	All	–	–	–	$\mathfrak{n}/2$	Sync	A–w–A
[DPSZ12, CDE+18]	All	$\mathfrak{n}$	Synch	A–w–A	$\mathfrak{n}$	Sync	A–w–A

Table 1: Summary of Protocol Choices and Their Benefits. Here the protocol name refers to a suitably obvious generalization of the protocol specified. The value A–w–A means Active–with–Abort.

4.3.5.1 Using an Active–with–Abort Offline Phase: It is relatively straight forward to replace our (robust) offline protocol with (for example) the one marked Protocol 1 from [JSvL22]; which provides active–with–abort security. The offline Protocol 1 from [JSvL22] uses a passively secure multiplication protocols, essentially "Maurer multiplication"[2], and these are then checked using the Offline 2 protocol from [JSvL22], which adapts the checking procedure from [KOS16, CDE+18]. The security of the offline phase of Protocol 1 of [JSvL22] is in the regime $t < n/2$, and it is the offline phase which has active–with–abort security. The online phase is by default robustly secure when $t < n/3$ as it is the same as our online phase.

The paper [JSvL22] looks at $q = 2^k$ only, but extending this to $q = p_1 \cdots p_k$ is easy to do. The paper also considers what we call the *nSmall* regime (as they use PRSS operations extensively in the offline phase). The use of PRSS could be removed for the *nLarge* regime, at some minor cost in performance. The need for a complex dispute resolution framework is not needed, as the offline phase of Protocol 1 of [JSvL22] gives only active–with–abort security.

4.3.5.2 Using an Active–with–Abort Online Phase: We could also achieve $t < n/2$ in the online phase, in an active–with–abort security model with a synchronous network assumption, by replacing the robust reconstruction in our method *RobustOpen* with an abort when an error is detected. However, **our** offline phase would only provide security for $t < n/3$ due to the need to open a sharing of degree $2 \cdot t$ with error-detection; thus a different offline phase (such as that alluded to above) would be needed. We will not pursue these option further in this document however. The alterations needed to achieve $t < n/2$ and active–with–abort security in the online phase are obvious. This leads to the decision

> **Design Decision 11: An Honest Majority Option**
>
> When no offline phase is needed our protocols can operate in the $t < n/2$ regime in an active–with–abort MPC model assuming a synchronous online phase. We will not discuss this further in the document, except to say it is easily enabled with obvious changes.

4.3.5.3 Using a Monolithic Active–with–Abort Protocol: An alternative to our MPC protocols would be to not work in an offline/online paradigm. In which case one can obtain, in the case of active–with–abort security (and $t < n/2$), protocols which are around 50% more efficient (if one adds the cost of the online and offline together), but are around twice as expensive (if one only compared the online costs) compared to say Protocol 1 from [JSvL22]. Here the most efficient solution would be to adapt the protocol of Chida et al. [CGH+18], which was presented as Protocol 5 in [JSvL22] for the case of Galois Rings.

4.3.5.4 Using Full Threshold Active–with–Abort Protocol: A final alternative would be to utilize the SPDZ and SPDZ2k protocols [DPSZ12, CDE+18]. These provide, over synchronous networks, an active–with–abort secure protocol, which is secure for $t < n$. Extending the SPDZ/SPDZ2k protocols to work for $q = p_1 \cdots p_k$ is relatively straight forward. However, this protocol is very inefficient, compared to ones which are not full threshold, if one takes into account the amount of time needed to execute the offline phase.

4.3.5.5 Mixed Adversary MPC and MPC with Frieds and Foes: As we are dealing with robust MPC there is a "quirk" of the MPC security model which we feel should be discussed. The MPC security model says nothing about the privacy of the data between the honest parties, after all the truly honest parties are not even considered to be curious. In other words if honest parties obtain the private information of other honest parties it is not considered a privacy breach.

---

[2]By which we mean the classical multiplication method of Schur-Multiplication, Reshare, Recombine.

This results in the following two related considerations:

1. An active adversary which controls t parties could send all the secret shares they hold to one (or more) of the honest parties. The receiving honest party would now have $t+1$ shares and, if a Shamir sharing of degree t was being used, they could then reconstruct all of the secret data. This is not considered a privacy breach in the normal MPC security model, as it is an honest player who obtains the data and not the adversary. In the discussion which follows we refer to this as an *adversarial privacy breach*.

2. On the other hand many robust MPC protocols, especially those using the player elimination framework or the dispute resolution framework, work by identifying bad players and then removing them from the computation. This could be by explicitly removing them from the protocol by reducing n and the associated degree t of the Shamir sharing, or by implicitly removing them by setting their share values to zero (as is done in [DN07]. Indeed some protocols, for example [BGIN19], proceed by open all remaining secret shared values to the honest players, once all bad players have been identified. The honest players then compute the final result "in the clear". In both cases this results in the honest players obtaining some, or all, information held in the secret shares, and thus *could* be considered a privacy breach; although it again is not an privacy breach in the normal MPC model. In the discussion which follows we refer to this as a *protocol privacy breach*.

Note, one can consider a protocol which exhibits protocol privacy breaches as enabling a form of adversarial privacy breach, in which the adversarial transfer of information from the active adversarial parties to semi–honest adversarial parties is done via the protocol as opposed to being done directly. Thus for many protocols, such as the two mentioned [BGIN19, DN07], the two notions are equivalent.

These issues have been known since the early days of MPC. Indeed, it is discussed in [RB89], where a "remedy" of running the a protocol using a Shamir sharing of degree $t+1$ to protect against t active adversaries is suggested. This increases the minimum number of parties (as a function of the bound on the number of adversarial parties t) needed to execute the protocol. It turns out such an increase in the number of parties is unavoidable, as we shall now recap.

Over time these forms of privacy breach have resulted in the literature creating two related but distinct forms of security model: A model called the mixed adversary model, which was introdoced in [FHM99], and a model called MPC with Friends and Foes, which as introduced in [AOP20]. Both models are parameterized by two values t_a and t_h. The value t_a indicates how many adversarial parties are (statically) actively corrupted, whilst t_h indicates how many parties are (statically) corrupted in an honest–but–curious fashion.

In the mixed adversarial model there is a single monolithic adversary controlling the $t_a + t_h$ bad parties, resulting in a notion we shall denote by (t_a, t_h)–Mixed security. In the MPC with Friends and Foes model there are two distinct adversaries, one $\mathcal{A}_a$ controlling the t_a active parties and one $\mathcal{A}_h$ controling the t_h semi–honest parties, a notion we shall denote by (t_a, t_h)–FaF security. Technically, to capture the adversarial privacy breach mentioned above the (t_a, t_h)–FaF security model requries the simulator for $\mathcal{A}_h$ to obtain the ideal world view of the simulator for $\mathcal{A}_a$.

Note that (t_a, t_h)–Mixed security neither implies, nor is implied by, (t_a, t_h)–FaF security. Thus the two notions are incomparable. However, both $(t_a, 0)$–Mixed security and $(t_a, 0)$–FaF security are the standard notions of robust MPC against t_a active adversaries.

The limitations of (t_a, t_h)–FaF security, and some constructions, have also been considered in [ABO23, MRY23]. There has also been a consideration of potentially practical (t_a, t_h)–FaF for ring–based MPC for small values of t_a, t_h and n in works such as [HKK$^+$22, KKPG22].

The main problem with dealing with (t_a, t_h)–Mixed and (t_a, t_h)–FaF security is that these models imply larger lower bounds on the number of parties; after all we have more adversaries so to counter this we require most truly honest parties. See Table 2 for a quick summary of these lower bounds on n.

	Normal MPC Active Security	(t_a, t_h)–Mixed/FaF Security

Computational MPC	$\mathfrak{n} > 2 \cdot \mathfrak{t}_a$	$\mathfrak{n} > 2 \cdot \mathfrak{t}_a + \mathfrak{t}_h$
Statistical MPC with Broadcast Channel	$\mathfrak{n} > 2 \cdot \mathfrak{t}_a$	$\mathfrak{n} > 2 \cdot (\mathfrak{t}_a + \mathfrak{t}_h)$
Statistical MPC with no Broadcast Channel	$\mathfrak{n} > 3 \cdot \mathfrak{t}_a$	$\mathfrak{n} > \max(2 \cdot (\mathfrak{t}_a + \mathfrak{t}_h), 3 \cdot \mathfrak{t}_a)$
Perfect MPC with Broadcast Channel	$\mathfrak{n} > 3 \cdot \mathfrak{t}_a$	$\mathfrak{n} > 3 \cdot \mathfrak{t}_a + 2 \cdot \mathfrak{t}_h$

Table 2: Lower bounds on $\mathfrak{n}$ for $(\mathfrak{t}_a, \mathfrak{t}_h)$–Mixed and $(\mathfrak{t}_a, \mathfrak{t}_h)$–FaF security compared to normal MPC security, i.e. $(\mathfrak{t}_a, 0)$ security.

This leads us to the following design decision related to these notions; as we want to be able to support a small number of parties we focus on the standard MPC active security definition. However, system implementors need to be aware that the adversarial privacy breach mentioned above is therefore always possible.

Design Decision 12: $(\mathfrak{t}_a, \mathfrak{t}_h)$–Mixed/FaF Security

We do not consider the notiona of $(\mathfrak{t}_a, \mathfrak{t}_h)$–Mixed/FaF security in this document due the larger implied lower bounds on the number of parties $\mathfrak{n}$.

4.3.6 MPC–Based Bit Generation

The main use of our MPC protocols is to generate shared random bits. We do these via classical/folk–lore methods which we describe in Section 7.1.5 and Section 7.4.1. The goal is to produce a shared bit modulo q.

The method for **power–of–two** q appeared first in [OSV20], however we use a simplification which first appeared in [DDE$^+$23]. The work of [OSV20] considers bit generation only in $\mathbb{Z}/(q)$, however it is trivial to see that the method also applies in the more general context of Galois Rings.

Generating bits when $q = p_1 \cdots p_k$ is a product of primes is more complex. The method for odd **prime** q we think first appeared in the MPC context in [DKL$^+$13]. When q is a product of odd primes then the usual method (usually denoted by dabit/mabit generation) is to generate shared bit in k MPC engines (one for each prime divisor p_i). As we are using Shamir Sharing modulo q this is equivalent to finding a shared bit modulo q via the CRT. The problem is that whilst it is easy to generate a bit modulo p_i, it is more tricky to generate **the same bit** modulo each p_i.

Prior work, based on dabits and mabits e.g. in [RST$^+$22, RW19, EGK$^+$20], are in the security model of active–with–abort; whereas we are targeting fully robust security. Thus it appears that the methods do not directly apply to our situation without significant changes. However, in the case of our *nSmall* threshold profiles there is a simple trick to obtain shared bits using the PRSS functionality. Finding a suitably **efficient** robust bit generation method for when $q = p_1 \cdots p_k$ in the *nLarge* setting we leave as an open research question.

Design Decision 13: Shared Bits Modulo $q = p_1 \cdots p_k$

In the case where $q = p_1 \cdots p_k$ we only present a method for shared bit generation in the threshold profile *nSmall*.

This decision creates the following knock on issue, that we do not deal with BGV and BFV at all in the threshold profiles *nLarge*.

Design Decision 14: Threshold Profiles for BGV and BFV

We only present threshold key generation and decryption for BGV and BFV in the *nSmall* threshold profiles.

The method for bit generation when $q = p_1 \cdots p_k$ requires us to make an assumption on the largest prime divisor p_L of q, namely that it is "suitably" large. This is to enable a mask to be applied to a shared bit modulo p_L, with the mask being large enough to mask the shared bit, but small enough not to wrap around modulo p_L. In our applications such an assumption will be automatically satisfied, so it is not really a constraint, but worth keeping in the back of ones mind when reading further.

> **Parameter Choice 2: Largest Prime Factor of** $q = p_1 \cdots p_k$
>
> In the case of our MPC protocols for threshold profile *nSmall*, and the case where $q = p_1 \cdots p_k$, we assume that the largest prime factor p_L of q satisfies $p_L > nSmallBnd \cdot 2^{stat+3}$.

Traditionally in the MPC literature the generation of bits has been considered part of the offline phase. For our MPC protocol we consider bit generation as part of the online phase. This is because our bit generation methods are secure over asynchronous networks, and not the synchronous networks we assume for the offline phase. Obviously this is a purely conceptual choice, and an implementation may choose to consider bit generation as part of the offline phase in terms of system construction. Indeed for the threshold decryption and key generation it makes more sense to consider bit generation as part of the offline phase.

> **Design Decision 15: Bit Generation in the Online Phase**
>
> We consider bit generation as part of the online phase due to the underlying networking assumptions when considering the MPC protocol itself. When considering the threshold key generation and decryption protocols later it may make more sense to think of bit generation as part of the offline phase.

4.3.7   MPC Protocols for Bit Decomposition

When using TFHE we can execute a high round MPC protocol for message extraction, which avoids the noise flooding approach discussed below. To implement this variant we utilize some standard generic MPC routines for bit-manipulation within an MPC engine. These algorithms are based on the methods in the documents [CdH10], [DFK$^+$06], [NO07] and [sec09]. Our bit decomposition method is information theoretically secure (i.e. the method itself reveals no more information than the information revealed by the underlying MPC system). Thus we avoid the statistically secure methods from the literature. We also utilize logarithmic round protocols for these tasks, as opposed to the (in practice) more expensive constant round protocols.

4.4   Fully Homomorphic Encryption

We describe in this document very simple variants of the BGV, BFV and TFHE encryption schemes. These descriptions are purely given to enable our threshold key generation and decryption to be described fully. We do not claim that our description of these conventional (i.e. non-threshold) schemes are efficient in terms of the homomorphic operations. There are many optimizations to all schemes which can be applied, which we do not discuss in this document.

In this document our FHE schemes will have plaintext modulus P and ciphertext modulus Q. This is to distinguish the moduli used in the FHE scheme (i.e. P and Q) from the modulus used in the MPC scheme (i.e. q). Sometimes we will have $q = Q$, but not always.

4.4.1 BGV

We present a variant of the BGV scheme [BGV12], which follows the description in [GHS12, HS20].
We present a leveled version of the scheme, which does not support arbitrary depth computations.
This is because bootstrapping for BGV is relatively slow; and we do not need to present it to describe
our threshold protocols. An implementation can add bootstrapping functionality to the scheme in the
normal way if desired.

   We touch upon specific optimizations for the BGV scheme, such as the use of Number Theoretic
Transforms (NTTs) via the Double–CRT representation (DCRT) which is common in most implementations;
but only as a methodology to implement the underlying ring multiplication efficiently. We do not cover
optimizations, such as keeping some levels of the DCRT representation in the NTT domain and some
levels in the coefficient domain. Again this is because this is orthogonal to what is needed to describe
our threshold protocols.

   As is standard we assume a form of circular security, in that we allow a ciphertext to encrypt bits of
the underlying secret key. This assumption can be removed, by utilizing a chain of public keys (one per
level), if desired. Assuming circular security enables us to provide a more concise presentation of the
scheme.

   The main divergence from some treatments, is that we utilize the *NewHope* noise distribution for our
LWE parameters. By the *NewHope* noise distribution we mean the following approximation to a Discrete
Gaussian Distribution, parameterized by a value $B \in \mathbb{N}$. On calling $NewHope(1, B)$ a set of $2 \cdot B$ random
bits $(b_i, b_i')_{i=1}^{B}$ are selected and then the value

$$s = \sum_{i=1}^{B} b_i - b_i'$$

is computed and returned. This produces a random variable with mean zero and standard deviation
$\sigma_B = \sqrt{B/2}$. When called with the syntax $NewHope(N, B)$, a vector of N such random variables are
produced. This distribution is particularly useful in our situation as BGV requires noise distributions
with small standard deviation, and such a distribution is easy to generate inside our basic MPC engine.
The $NewHope(N, B)$ noise distribution in full generality seems to have been first used in the context of
general LWE encryption in [ADPS16], although $NewHope(N, 1)$ was used for the noise distribution in
the BGV description given in [GHS12].

   A major divergence, in terms of performance, of our presentation of BGV compared to that in say
[GHS12, HS20], comes from us sampling a secret key for BGV from the distribution $NewHope(N, 1)$.
This results in larger bounds on the noise in the error analysis than given in [GHS12] and [HS20]. The
main difference being the noise formulae depend on N in our treatment and not $\sqrt{N}$. The reason for this
increase, from $\sqrt{N}$ to N, is due to the way the secret key is generated: In [GHS12] it is chosen to be
of fixed Hamming Weight, whilst in [HS20] it is chosen by rejection sampling until something of small
canonical norm is selected. The reason for this divergence is that generating secret shared samples of
the distribution $NewHope(N, 1)$ is relatively simple; whereas generating secret shared samples from
the distributions used in [GHS12] and [HS20] is relatively hard. Thus to simplify our threshold key
generation for BGV we pay a cost in terms of the parameters of the resulting BGV scheme.

   In summary:

4.4.2 BFV

We present a variant of the BFV scheme [FV12, Bra12]. The BGV and BFV schemes are closely related. The most important difference is their plaintext encoding: BGV encodes the plaintext in the "least significant bits" of the ciphertext whereas BFV encodes the plaintext in the "most significant bits". There are two ways one can deal with this difference. The first is to define BFV analogues of *all* the routines for the BGV scheme. The second is to apply a conversion routine from BFV to BGV, and vice–versa, in order that one can apply homomorphic operations to BFV style ciphertexts via the BGV routines. This conversion routine, which works when the plaintext and ciphertext moduli are co–prime, was first given in [AP13].

The second approach, via ciphertext conversion, results in identical noise growth for both schemes. A typical BFV application could then be implemented as in Figure 2. It is this second approach which we adopt for ease of exposition, and in addition because BGV has simpler noise formulae in the case of large P, when compared to BFV [CS16].
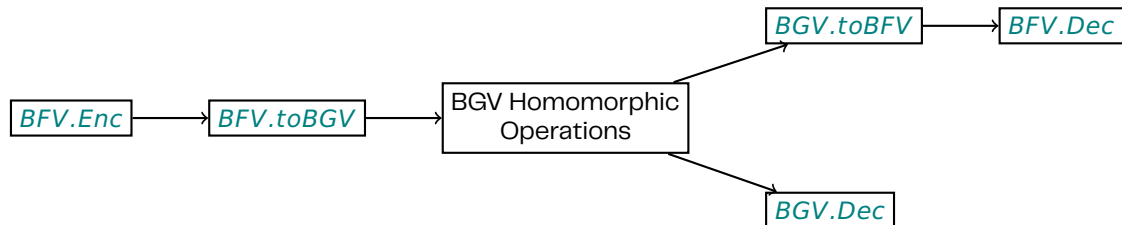
Figure 2: Typical execution path of BFV operations.

One could ask then, why would one want to even consider BFV as a separate scheme? Why would one then not use BGV all the time? This is a good question, which we leave to implementers. There may be some advantages in using a BFV encryption procedure which are specific to an application.

Given this second approach, we make exactly the same choices as we did above for BGV, for exactly the same reasons. In particular:

> **Design Decision 18: Design Choice for BFV**
>
> For ease of exposition we present a BFV scheme which
>   - Is leveled and does not support bootstrapping.
>   - Requires a circular security assumption.
>   - Various implementation optimizations (such as detailed manipulation of the DCRT represen–tation) are not explained.
>   - Noise distributions come from the noise distribution $NewHope(N, B)$.
>   - The secret key is chosen according to the distribution $NewHope(N, 1)$.

4.4.3 TFHE

The TFHE scheme [CGGI16, CGGI20, CLOT21, BdBB+25] has very different properties from BGV and BFV. To perform any homomorphic operations one needs access to a bootstrapping functionality. On the other hand bootstrapping in TFHE is very efficient and it enables arbitrary look–up tables to be computed during the execution of the bootstrapping operation (so called programmable bootstrapping). Indeed our main low–round threshold decryption protocol will use a variant of the bootstrapping operation.

    We describe a variant of TFHE which assumes a circular security assumption. The method of TFHE public key encryption that we employ is that described in [Joy24], which utilizes a Ring–LWE instance in order to generate many (standard) LWE instances. This avoids the need to have a public key which contains a large number of encryptions of zero, allowing for more compact public keys. It is also conceptually related to the public key encryption methods used in BGV and BFV.

    However, the resulting LWE dimension of the output ciphertext of public key encryption is $\hat{\ell}$, which is required to be a power of two. The internal LWE ciphertexts for the main homomorphic algorithms of TFHE can be made more efficient if they are done with a dimension ℓ which is not a power of two; and potentially smaller. Thus we introduce a key–switch from the LWE dimension $\hat{\ell}$ to an LWE dimension ℓ after the application of the public key encryption methodology of [Joy24]. We will call this specific key–switch a dimension–switch later, to avoid confusion with the more standard use of key–switching in TFHE.

    A key aspect, which makes TFHE bootstrapping so fast, is the fact that TFHE ciphertexts are relatively small. In particular the LWE dimension and ciphertext modulus are relatively small. This means that noise distributions for the underlying LWE problems can no longer have small standard deviations, in order to ensure security via the lattice–estimator.

    To ensure we can generate the noise distributions efficiently within our MPC system, for threshold key generation, we (as we did for BGV and BFV) avoid the use of discrete Gaussians as the noise distribution. However, even the distribution $NewHope(N, B)$ is too costly, due to the large standard deviations needed for TFHE keys and encryption. Instead we use for the noise distributions a "tweaked" uniform distribution on the interval $[-2^b, \ldots, 2^b]$ for some integer value of b; which we shall denote by $TUniform(1, -2^b, 2^b)$. When we call $TUniform(N, -2^b, 2^b)$ we obtain a vector of N such random variables. This change in distribution makes very little difference to the overall security analysis of TFHE, as experiments with the lattice estimator confirm.

    The distribution $TUniform(1, -2^b, 2^b)$ is defined as follows, any value in the interval $(-2^b, \ldots, 2^b)$ is selected with probability $1/2^{b+1}$, with the two end points -2^b and 2^b being selected with probability $1/2^{b+2}$. A value from this distribution can be selected by drawing $b + 2$ uniformly random bits $b_0, \ldots, b_{b+1}$, and then computing the value

$$x = \left(\sum_{i=0}^{b} b_i \cdot 2^i \right) - 2^b + b_{b+1}.$$

Thus the resulting variable has mean zero, and variance $(2^{2 \cdot b+1} + 1)/6$. This last fact can be established

35

by writing

$$Var(X) = \sum_{i=0}^{b} 2^{2i} \cdot Var(B_i) + Var(B_{b+1})$$

$$= \frac{1}{4} \cdot \left(1 + \sum_{i=0}^{b} (2^2)^i \right) = \frac{1}{4} \cdot \left(1 + \frac{1 - (2^2)^{b+1}}{1 - 2^2} \right)$$

$$= \frac{1}{4} \cdot \left(1 - \frac{1}{3} + \frac{(2^2)^{b+1}}{3} \right) = \frac{1}{6} + \frac{2^{2 \cdot b}}{3}$$

where B_i are independent uniform random variables on $\{0, 1\}$.

In summary

Design Decision 19: Design Choice for TFHE

For ease of exposition we present a TFHE scheme which
- Has plaintext and ciphertext moduli a power of two.
- Supports programmable bootstrapping.
- Requires a circular security assumption.
- Various implementation optimizations are not explained; however the use of FFT is (as it affects parameter sizes and enables efficient ring multiplication).
- Public key encryption is via Ring–LWE, as explained in [Joy24]. However, to enable efficient parameters to be generated we follow it by a dimension switch to potentially smaller LWE dimension.
- Noise distributions come from the noise distribution $TUniform(N, -2^b, 2^b)$.

4.5 Broadcast

We remarked above that broadcast channels need, in practice, to be implemented over the point–to–point channels which the basic internet protocols provide us with.

A reliable broadcast protocol is one which allows a sender S to broadcast a message m to a set of parties with the following guarantees:
- If one honest party terminates, then every honest party eventually terminates.
- If S is honest then every honest party eventually terminates.
- Any two honest parties that terminate output the same message, and if S is honest then that message is m.

Note, if S is dishonest then the protocol can possibly never terminate in an asynchronous network.

Over asynchronous networks one can utilize the broadcast protocol of Bracha [Bra87], which works when $\mathfrak{t} < \mathfrak{n}/3$. In this protocol the parties are assumed to be connected by authentic channels. However, our asynchronous protocols never make use of a broadcast operation. But our synchronous protocols will require a reliable broadcast mechanism. In a synchronous network we assume that every round has a time–out value, Δ. Thus *if* a party sends a message in this round, *then* it *will be received* within Δ "seconds", by the receiving party. For synchronous networks we can simplify Bracha's protocol slightly[3]. The synchronous version is also reliable when $\mathfrak{t} < \mathfrak{n}/3$, however with the following enhanced guarantees
- Every honest party will terminate will either a message m, or a value $\perp$.
- Any two honest parties that terminate output the same value, and if S is honest then that message is m.

One could think that it might be more efficient to implement the Dolev–Strong protocol [DS83]. One major advantage of using the Dolev–Strong protocol would be that it can tolerate up to $\mathfrak{t} < \mathfrak{n}$ adversaries.

---

[3]See https://hackmd.io/@alxiong/bracha-broadcast for a discussion of this.

In addition Dolev–Strong requires $\mathfrak{t} + 1$ rounds, whereas our protocol requires $\mathfrak{t} + 4$ rounds. However, the main cost of Dolev–Strong is the need to generate and verify digital signatures; and indeed chains of such signatures. Such chains can be expensive to verify, especially for large values of $\mathfrak{n}$, or if one wants to use post–quantum primitives. Since our MPC protocols only tolerate up to $\mathfrak{t} < \mathfrak{n}/3$ adversaries we prefer the modified version of Bracha broadcast.

The broadcasts we perform will be within our underlying MPC protocol, which we aim to base solely on lightweight cryptographic primitives such as symmetric key encryption, MACs and hash functions (except for a set–up phase in which any pairwise secret keys are distributed amongst the parties).

> **Design Decision 20: Choice of Synchronous Broadcast Mechanism**
>
> We utilize a variant of Bracha broadcast when executing a broadcast in order to avoid the need to verify chains of digital signatures.

4.6   Reed–Solomon Codes and Shamir Secret Sharing over Galois Rings

To deal with threshold decryption of FHE schemes in which the ciphertext modulus is not a prime, but a power of a prime $q = p^k$ or a product of primes $q = p_1 \cdots p_k$, we need to deal with LSSS–based MPC, and hence Shamir Secret Sharing, and hence Reed–Solomon codes over Galois Rings. The main source we take for most of our exposition is that in the thesis [Feh98] and the paper [ACD$^+$19]. The reader is also referred to [QBC13] for a treatment which also includes the examination of Reed–Solomon codes over more general (even non–commutative) rings.

In almost all prior works on Reed–Solomon codes (and Shamir Secret Sharing) over Galois Rings, the underlying ring is a ring extension of $\mathbb{Z}/(p^k)$. Reed–Solomon codes, and hence Shamir Secret Sharing, over $\mathbb{Z}/(q)$ for $q = p_1 \cdots p_k$ follow immediately from the Chinese Remainder Theorem (CRT) from the equivalent codes over $\mathbb{Z}/(p_i)$, but they do not seem to appear in generality in the literature.

Note we deal with $q = p^k$ and $q = p_1 \ldots p_k$ using sometimes distinct algorithms (Hensel lifting and the CRT) we could combine these two methods together for a more general q of the form $q = p_1^{k_1} \cdots p_t^{k_t}$; but this added complication is not needed for our applications

> **Design Decision 21: Choice of q in MPC Protocols**
>
> We decide to describe our secret sharing, and hence MPC protocols, only in the context of $q = p^k$ a power of a prime, and $q = p_1 \cdots p_k$ a product of primes. One could present the protocols in a more general context of $q = p_1^{k_1} \cdots p_t^{k_t}$ with little added complication. We choose not to, as we do not need this generality, and in addition we do not want to confuse the reader too much.

We utilize a ring extension
$$GR(q, F) = (\mathbb{Z}/(q))[X]/F(X)$$

where $F(X)$ is a degree $\mathfrak{d}$ polynomial in $\mathbb{Z}[X]$ which is irreducible modulo each prime divisor of q. The need for a ring extension of $\mathbb{Z}/(q)$, i.e. some ring of size $q^{\mathfrak{d}}$, is to enable enough values to exist in order to be able to define secret sharing amongst $\mathfrak{n}$ players. In particular we impose the following constraint:

> **Parameter Choice 3: Size of $\mathfrak{d}$ (1)**
>
> If p is the smallest prime dividing q then we pick $\mathfrak{d}$ such that $p^{\mathfrak{d}} > \mathfrak{n}$.

Our description of the Berlekamp–Welch algorithm for fields is the standard one [BW86]. The discussion as to how this is Hensel lifted to Galois Rings over $\mathbb{Z}/(p^k)$ follows from the method in [ACD$^+$19]. The method to lift via the CRT for when $q = p_1 \cdots p_k$ is immediate. The paper [ACD$^+$19] essentially tells

us that all the properties of Shamir sharing we use in MPC protocols over fields, also carry over to Galois Rings. Including such techniques such as randomness extraction, and robust opening.

Our description of Syndrome Decoding for "traditional view" Reed–Solomon codes we take from a set of lecture notes [Hal15]; indeed we could find little explanation of this elsewhere in down–to–earth terms. In [Hal15] Syndrome Decoding for traditional view Reed–Solomon is presented for the finite field case. Extending this to the Galois Ring case considered in this document is relatively simple. Our presentation can be seen as a simplification, to the specific case we are interested in, of the more general presentation of Syndrome Decoding for Galois Rings presented in [GTLBNG21].

At many points in protocols we would like to generate a degree $\mathfrak{t}$ Shamir sharing of a random value. When $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "small" (which, recall, we deem to mean less than *nSmallBnd*) we utilize a PRSS. This is a technique which goes back to [CDI05]. The benefit of a PRSS is that it generates random values non–interactively, however the overall computational complexity depends on $\binom{\mathfrak{n}}{\mathfrak{t}}$, which can become exponentially big as $\mathfrak{n}$ increases. The same construction of a PRSS allows us to construct a Pseudo–Random Zero Sharing (PRZS). This is similar to a PRSS, except now the sharing is guaranteed to be of the zero value, and the sharing is of degree $2 \cdot \mathfrak{t}$, and not $\mathfrak{t}$. We also utilize a PRSS to generate shares of random values of bounded sizes in our protocols; this method enables us to mask "small" values within our MPC engine and then open them. These are used in a number of places, to simplify protocols.

Due to their exponential asymptotic complexity we only utilize a PRSS and PRZS in threshold profiles where $\binom{\mathfrak{n}}{\mathfrak{t}}$ is small, i.e. the profiles which we called *nSmall*. When $\binom{\mathfrak{n}}{\mathfrak{t}}$ is large we need to utilize more complex primitives, and hence more complex protocols.

In the case where $\binom{\mathfrak{n}}{\mathfrak{t}}$ is small we provide three different initialization routines for our PRSS: One which is passively secure, one which is active–with–abort secure, and one which is fully robustly secure. The robust initialization routine we could also not find in the literature, but again it is obvious and trivial.

An advantage of PRSS methods is that they enable the generation of secret shared small elements within the Galois Ring. The interactive equivalenet of PRSS is essentially the Verifiable Secret Sharing (VSS) methods of the next section. For VSS methods it appears hard to generate small values; a fact which will add to our protocol complexity when $\binom{\mathfrak{n}}{\mathfrak{t}}$ is large.

4.7   Verifiable Secret Sharing

When $\binom{\mathfrak{n}}{\mathfrak{t}}$ is large, i.e. in the threshold profiles we are denoting *nLarge*, the above PRSS method to generate random elements is not available. Thus we need a method of interactively generating a shared random element in the Galois Ring.

We utilize two forms of verifiable secret sharing, when working with the threshold profiles in category *nLarge*. The first is a traditional VSS protocol, which is perfectly secure. This first variant is only used rarely, it is used to execute a *CoinFlip* protocol in a robust manner to generate random challenges for our second VSS protocol. Thus for this protocol we utilize the four round VSS protocol from [GIKR01], although we take the description from [CCP22] (where it is labeled as Protocol 4GIKR-VSS-Sh). This is chosen as it is information theoretically secure, has a relatively small number of rounds, whilst still being in the regime of $\mathfrak{t} < \mathfrak{n}/3$. To ensure that the value produced from this VSS execution has enough entropy for the *CoinFlip* we require that $q^{\mathfrak{d}} > 2^{sec}$. If the specific q being used for the rest of our protocols is not this large, then one could just execute this first VSS using a temporary larger value of q.

> **Parameter Choice 4: Size of $\mathfrak{d}$ (2)**
>
> We pick $\mathfrak{d}$ such that $q^{\mathfrak{d}} > 2^{sec}$, or temporarily increase the size of q within the protocols for the VSS and *CoinFlip*.

The second VSS is the one we actually use to generate random sharings in the threshold profiles in category *nLarge*. It replaces the PRSS/PRZS operations used for the threshold profiles in category

nSmall for generating random (but not small) secret shared values in the Galois Ring. This second VSS is a statistically secure batched VSS procedure, which is the natural translation to the Galois Ring setting of the equivalent finite field based protocol from [DN07]. The key advantage of this protocol is that it produces a large number of valid sharings in a one-shot manner.

4.8 Threshold FHE as a Basis for MPC

Threshold FHE has been considered quite a bit in the literature in the context of enabling MPC protocols. Indeed, Gentry in his thesis [Gen09] provides a simple Secure-Function Evaluation (SFE) protocol in which parties use an FHE algorithm to encrypt their inputs to each other, the parties then compute independently the function output homomorphically, and then the (public) output is obtained via a threshold decryption protocol. One can view the threshold decryption protocol in Gentry's SFE protocol above as a mini-MPC protocol, thus we have (in some sense) built an SFE protocol from an FHE scheme and a mini-MPC protocol, with the key generation itself performed by another mini-MPC protocol.

MPC and FHE have been combined in other ways. For example the SPDZ protocol [DPSZ12] produces an actively-secure full threshold MPC protocol based on LSSS technology, but one which uses an FHE scheme supporting circuits of multiplicative depth one (so called Somewhat Homomorphic Encryption (SHE) schemes of depth one) as a means of providing an efficient offline phase. Indeed one can view the offline phase of SPDZ as a variant of Gentry's FHE-based MPC protocol for the functionality of producing Beaver multiplication triples. To obtain active security in SPDZ one needs to augment the FHE ciphertexts with Zero-Knowledge Proofs of Knowledge (ZKPoKs). There are ZKPoKs which have been specifically designed for this usage in the SPDZ protocol [BCS19]. We decide not to target full threshold in our document as we are focused on robust protocols, which are impossible in the full threshold environment.

In a different direction, [CLO+13] provides a *robust* MPC protocol, in the honest majority setting, which utilizes an SHE scheme of a specified depth, which follows Gentry's blueprint for SFE. The extension from SHE to supporting any function is enabled by replacing the bootstrapping in FHE by a special protocol based on distributed decryption; namely bootstrapping is performed by interaction. It is the basic threshold approach of [CLO+13] which we will follow in this document for threshold profiles *nSmall*.

4.9 Threshold FHE Key Generation

The above works on using FHE to obtain MPC all follow the approach to key generation that we follow. Namely they execute the threshold key generation via a generic MPC protocol. This may seem overkill, however, as was pointed out in [RST+22] for the case of threshold key generation for the SPDZ protocol, the key generation algorithms for FHE are particularly suited to MPC evaluation. Apart from the need to generate "small elements" (which can be done via a random bit-sampling methodology) the entire key generation algorithm is almost all linear. Thus almost all of the complex key generation can be performed locally, and requires no interaction. Since FFT/NTT algorithms are also linear one can even execute the "optimized" versions of such key generation algorithms in the FFT/NTT domain locally as well.

The advantage of the generic MPC approach is that the output of the key generation methodology is exactly the same as if the key generation was executed by a trusted third party. For FHE key generation this is particularly important as it means that noise distributions, which affect the performance and evaluation potential of the final FHE scheme, are not impacted (too much) by the need to generate the keys in a threshold manner. Thus (for example) the number of players has very little affect on the parameter sizes being chosen. Indeed the only affect of the number of players is essentially on the ability to implement the underlying MPC protocol in the first place.

> **Design Decision 22: Threshold Key Generation via Generic MPC**
>
> We elect to implement threshold key generation via generic MPC in order to ensure FHE parameters are not adversely affected by the threshold requirement.

For BGV/BFV our generic key generation methodology results in parameters enabling threshold decryption which are roughly the same as for those requiring no threshold decryption capability[4]; however the number of levels which can homomorphically be evaluated decreases a little. For TFHE we find there is almost no difference in the parameters which support threshold key generation compared to those which do not.

The required number of secret shared random bits can be quite high if we adopt a key generation method which is compatible only with our desire to enable threshold decryption. Thus, in Appendix B we sketch a method which uses less shared random bits for the threshold profile *nSmall*. However, this method requires us to increase the FHE parameters by more than is strictly required just to enable threshold decryption.

There are other approaches (in particular MK-FHE and MP-FHE which we outline below), however these other approaches result in FHE schemes which are slightly different, or have slightly larger parameters, than those proposed in single party FHE. We emphasize that in our work we envisage the use of standard FHE scheme's with almost the same parameter sets for the single user case; but we use them in a multi-user environment. Thus all of the optimizations in existing FHE libraries for single user FHE schemes can be applied to applications using our threshold variants. Such an approach is vital for schemes such as TFHE, where the fast bootstrapping operation is only available due to the very small parameter sizes that are picked for the instantiation. For BGV this approach is not so important.

4.9.1 Multi-Key FHE

There are other approach to threshold key generation, of which the most popular is so-called Multi-Key FHE (or MK-FHE), see for example [AJL+12, AJW11, LTV12] amongst many other works. In these works, instead of parties generating a global single FHE key in an initialization phase (as one does using the generic MPC approach) the parties take their existing individual (multiple) FHE key pairs and combine them in order to perform an MPC-like computation. Whilst this provides a simpler operational setup, the practical implementations of MK-FHE are not as efficient as single key FHE.

The papers [AJW11, AJL+12, LTV12] introduce various LWE-based FHE schemes, which are key-homomorphic, specifically for the task of creating MK-FHE. The paper of Lopez-Alt et al [LTV12] based it's construction on an NTRU-like assumption, which was further extended in the YASHE FHE scheme [BLLN13]. However, such "overstretched" NTRU-based schemes were subsequently shown to be insecure [ABD16].

The key theoretical usage of MK-FHE is to produce very low round MPC protocols. For example if one allows the use of the Common Random String (CRS) model then MK-FHE enables two round actively secure MPC, [MW16]. A similar result [GGHR14] is possible, without using MK-FHE, but using instead Indistinguishability Obfuscation (iO). Assuming MK-FHE is, of course, theoretically nicer than assuming iO, since iO is not so well established a primitive. Two rounds is thus known to be both necessary and sufficient for MPC in the CRS model. In the plain model (i.e. without a setup assumption such as a CRS) it was for a long time an unknown problem as to how many rounds were both necessary and sufficient for actively secure MPC. In 2016 it was shown [GMPP16] that four rounds were necessary, and then in 2017, using MK-FHE, it was shown [BHP17] that four rounds were indeed sufficient for actively secure MPC. If one assumes a PKI then three rounds are possible, as shown (for semi-malicious adversaries) in [KLP18].

---

[4]Except for the increase in parameters due to the different secret key distribution which we remarked upon earlier

Further, theoretical, work has been carried out on MK–FHE–based MPC protocols. For example, Damgård et al [DPR16] develop a form of *equivocable* FHE, which enables them to extend the work of [AJW11, AJL$^+$12, LTV12] from the static corruption model to the adaptive model, whilst (again) giving a secure MPC protocol in the dishonest majority setting.

Despite over ten years of research very little progress has been made on turning MK–FHE into a practical reality, see for example [BP16, CCS19, CDKS19, CZW17, CM15, KKL$^+$23, AKÖ23, KMS24, LP19, PS16, YKHK18]. The extra functionality required by such MK–FHE schemes requires larger parameter sets, and more complex algorithms, than are needed in standard FHE schemes. For example, even the most efficient schemes require FHE scheme parameters which scale with the number of intended users.

4.9.2 Multi–Party FHE

Another variant of MK–FHE is that of Multi–Party FHE (MP–FHE). The work on MP–FHE is much like our own application; namely there is a distributed key generation protocol and a distributed decryption protocol. However in MP–FHE, the key derivation is usually created directly from the underlying mathematics of the FHE encryption scheme, as opposed to applying generic MPC technology. MP–FHE has also found application in theoretical cryptography. For example [Coh16] constructs constant round protocols for MPC over asynchronous networks, given an asynchronous Byzantine agreement functionality, assuming static malicious adversaries when $t < n/3$. In [CsW19] a two round, semi–honest, SFE protocol is given with sub–linear communication complexity.

4.9.3 Resharing

We also present a method for transferring a sharing from one set of parties to another, or even resharing a value within a given set. This enables us to achieve a form of pro–active security, in which we can reshare the underlying FHE secret key at given intervals of time.

This method is very similar to other methods for share re–randomization used in the MPC literature (which go back to [BGW88]). The basic idea is for the players in the source set to transfer a sharing of their shares to the players in the second set. The players in the second set then open a syndrome polynomial, and apply syndrome decoding in order to correct any invalid shares sent.

4.10 Threshold FHE Decryption

When using threshold decryption for FHE schemes, even when using noise flooding, one needs to worry about attack vectors related to correctness of the underlying FHE operations, so called IND–CPA^D attacks. Such attack vectors were initiated, for non–exact FHE scheme such as CKKS, in the work of [LM21] (although attacks related to correct evaluation for LWE–based systems go back to at least [DVV19]). In a series of papers in early 2024 these attacks were shown, in a series of papers [CSBB24, CCP$^+$24, ABMP24], to also apply to exact-FHE schemes (i.e. BGV, BFV and TFHE). In this document we set the error probability for FHE operations to be exponentially small (we choose an error probability of $err \approx 2^{-128}$ in this document), in order to render the above attack vectors computationally infeasible.

There are two approaches to threshold FHE decryption which we examine in this document: An approach using noise–flooding and an approach using generic MPC. The latter approach is only applicable to TFHE in this document. Extending the generic MPC approach to BGV and BFV is possible, but we feel not relevant given the efficiency of the noise–flooding approach for BGV and BFV.

4.10.1 Threshold Decryption via Noise–Flooding

At about the time of Gentry's thesis on FHE in 2009 [Gen09], the first threshold key generation and decryption for LWE based ciphertexts was given by Bendlin and Damgård [BD10]. Their methodology used replicated secret sharing to split the secret key, a method whose complexity scales with $\binom{n}{t}$. The simpler case of full–threshold, i.e. $t = n - 1$, decryption for LWE ciphertexts was combined with SHE and formed the basis of the SPDZ MPC protocol [DPSZ12]. This utilized the BGV encryption scheme, supporting circuits of multiplicative depth one, and used the noise flooding technique. The same techniques were then used in the context of FHE by Asharov et al [AJL$^+$12] in the full threshold setting. A similar application and usage was also given in [CLO$^+$13], which considered the threshold setting of $t < n/3$ via Shamir sharing. In our work we shall adopt the methodology of [CLO$^+$13] for our main threshold decryption protocol.

A generic thresholdizer for arbitrary protocols was given by Boneh et al. in [BGG$^+$18] using threshold–FHE. The construction of Boneh et al. utilizes a special form of secret sharing called $\{0, 1\}$–LSSS, which is closely related to replicated sharing; and thus does not scale to more than a few players.

All of these prior works utilized noise flooding as a methodology. As remarked above this requires a super–polynomial gap between the bound on the noise term e and the ciphertext modulus q. Such super–polynomial blow–ups in other areas of cryptography based on LWE have recently been avoided by utilizing the Renyi divergence [BLR$^+$18]. This, as an approach to threshold–FHE, was recently examined by [BS23] and [CSS$^+$22]. The problem with using the Renyi divergence in the context of distributed decryption, is that the general technique of Renyi divergence is hard to apply to security problems which are inherently about distinguishing one distribution from another. In [CSS$^+$22] and [BS23] a way around this was found by designing special security games for threshold–FHE usage, which enabled the use of the Renyi divergence. The problem is that these games need to cope with the homomorphic nature of the underlying encryption scheme, and thus cannot be adaptive. In applications we really require a threshold–FHE protocol which is indistinguishable, to an adversary, from a simulation interacting with an ideal functionality. The security games presented in [CSS$^+$22] and [BS23] do not allow such a usage.

> **Design Decision 23: Simulation Security vs Game Based Security**
>
> Our threshold key generation and decryption protocols are secure in the UC (simulation based) security model. This enables them to be arbitrarily composed in other protocols.

Simulation security for threshold LWE can be obtained without using noise flooding, as was described in [MS23]. However, their technique only can be applied to traditional LWE public key encryption schemes which do not allow any form of homomorphic operations. An interesting open research question is to obtain simulation secure threshold homomorphic decryption without the need for noise flooding.

Thus we are led back to considering noise flooding for which simple protocols are available which provide simulation based security. However, as detailed above, for FHE schemes such as BGV and BFV this is not a problem. The only issue comes with schemes such as TFHE, which utilize small parameters in order to achieve very fast bootstrapping operations.

We present a simple method for threshold decryption for TFHE ciphertexts in the presence of $t < n/3$ actively (but statically) corrupted adversarial parties. Our methodology produces a threshold decryption functionality which is in the simulation paradigm, this makes it more amenable to being used as a black box in larger protocols than the game–based approaches based on Renyi divergence.

Our method uses the approach of [DDE$^+$23], which is itself based on [CLO$^+$13], which works for arbitrary prime power values of q, including the important case of $q = 2^{64}$ for TFHE. Adapting it to the case of $q = p_1 \cdots p_k$, for use with BGV and BFV, is immediate via the Chinese-Remainder-Theorem. This method is based on the (relatively standard) trick of applying Shamir secret sharing over Galois Rings

mentioned above, thus the authors of [CLO$^+$13, DDE$^+$23] do not need to go via a replicated style secret sharing. In Shamir secret sharing the share sizes do not grow exponentially with the value of $\binom{n}{t}$.

When $\binom{n}{t}$ is "small" [DDE$^+$23] applies a trick, which first appeared in [CLO$^+$13], to enable threshold–FHE using a modified Pseudo-Random Secret Sharing (PRSS). In such a situation the protocol is a simple one round protocol, which is robust[5] and works over asynchronous networks when $t < n/3$.

When $t < n/2$, the authors of [DDE$^+$23] note that, they obtain a non–robust protocol, but one which has active–with–abort security. The proof of security in [CLO$^+$13] has a number of minor bugs/missing details in it, and is overly complex, thus [DDE$^+$23] also re–proves the main threshold–FHE result from that paper. It turns out that using two executions of a PRSS, for small values of $\binom{n}{t}$, automatically means we are adding a sum of at least two uniform distributions in the flooding term; and thus we can apply an improved statistical distance analysis in this case.

When $\binom{n}{t}$ is large the method of [DDE$^+$23] requires slightly more work; which is only sketched in [DDE$^+$23] but which we fully expand upon. In particular the threshold–FHE protocol is divided into two phases, an online and an offline phase. In the offline phase a "generic" MPC protocol is used to generate random shares of bits.

In the online phase, when $\binom{n}{t}$ is large, the threshold decryption protocol consume these random shares of bits to perform the threshold–FHE operation. In particular the shared random bits are used to produce two uniformly random noise flooding terms of the correct size[6]. Thus again, [DDE$^+$23] are able to apply the improved statistical distance analysis in this case.

The security properties of the offline phase are inherited from the underlying MPC protocol used to generate the shares of random bits. If the underlying MPC protocol is robust over asynchronous networks then so is the offline phase of our threshold–FHE protocol. If it only provides active–with–abort security over synchronous networks then they are the properties of our offline phase. In this document we describe an offline protocol which is robust, secure over synchronous protocols and supports $t < n/4$ when $\binom{n}{t}$ is large. Our online phase is again robust and works over asynchronous networks, when $t < n/3$; and is active–with–abort secure when $t < n/2$.

The methodology of [DDE$^+$23] for threshold–FHE decryption, for all values of $\binom{n}{t}$, follows in two conceptually simple steps:

1. We take the input ciphertext with LWE parameters (ℓ, Q) and then transform this (if needed) into a ciphertext with LWE parameters $(\overline{\ell}, \overline{Q})$ which encrypts the same message, with roughly the same noise bound in the two cases.

    (a) For BGV and BFV where Q is a product of primes this operation comes for free, by simply allowing threshold decryption to consume a couple of extra levels. In this case we utilize secret sharing/an MPC system with $q = Q$.

    (b) For TFHE Q is a prime power and $\overline{Q}$ is a prime power with $Q|\overline{Q}$. This switching to larger parameters is performed during a bootstrapping operation, which enables us to simultaneously reduce the noise, so that the noise gap is sufficiently large. We call this operation *SwitchSquash*, as it both switches the (ℓ, Q) values, and also squashes the noise. In this case we utilize secret sharing/an MPC system with $q = \overline{Q}$.

2. After enough noise–gap has been created, the above threshold decryption operation is performed via noise flooding.

For TFHE in practice the value Q will be 2^{64}, and we will only need to boost the modulus to a value of $\overline{Q} = 2^{128}$ in order to have a sufficient noise gap to perform threshold decryption. With such a value of $\overline{Q}$ it turns out that TFHE bootstrapping is still efficient, and thus the entire threshold decryption process is efficient. In particular it is very low round (requiring only one round in the online phase), thus it is often preferable to techniques based on generic MPC. In the special case when $\binom{n}{t}$ is small we obtain a

---

[5]i.e. it outputs the correct decryption even in the presence of malicious parties.
[6]As mentioned above bit generation can be considered as either an offline or an online procedure. As it is input independent, it makes more sense when thinking of threshold decryption to consider it as an offline procedure.

one-round, threshold decryption protocol which is robustly secure when $\mathfrak{t} < \mathfrak{n}/3$, with no offline phase, and which assumes only asynchronous, as opposed to synchronous, networks.

4.10.2 Threshold Decryption via Generic MPC

Another approach is to apply generic MPC to the problem of threshold decryption of FHE ciphertexts. This method is best suited for TFHE, as (as we remarked earlier) the noise–gap issue is not really an issue for BGV and BFV. In this method we take the sharing of the pre–decryption directly, i.e. we compute

$$\langle p \rangle = b - \mathbf{a} \cdot \langle \mathbf{s} \rangle.$$

One then needs to extract the message m from the pre–decryption. The message is encoded in p, for TFHE, in the following manner:

$$p = \Delta \cdot m + e \pmod{Q}.$$

We then extract the message m via bit–decomposition with the MPC engine (as for TFHE $\Delta = Q/P$ is also a power of two), thus only m is revealed and the value of e remains totally hidden. Thus, using Generic MPC, for a secret sharing system/MPC engine with $q = Q$, to perform the threshold decryption we do not need to require any noise gap. Hence, there is no need for any switch to larger parameters, or a large bootstrapping to produce a large noise gap.

   The disadvantage of this method is that we really need to apply the full power of our MPC engine. We also need the generation of random bits in order to perform the bit decomposition utilizing the standard techniques from the MPC literature mentioned above for bit manipulations. The use of such protocols means that threshold decryption requires a relatively large number of rounds (if $Q = 2^{64}$ as in TFHE, then this is about 16 rounds). Thus this method whilst not requiring the large bootstrap, may actually be slower, if the parties are separated by a network which requires a large ping–time.

   This method requires, for all values of $\binom{\mathfrak{n}}{\mathfrak{t}}$ an offline phase, which requires synchronous channels, as mentioned above. Thus the benefit of a purely asynchronous protocol, in threshold profile *nSmall*, is also lost by utilizing this second method. However, we feel there are a number of places where such a method could be useful (LAN networks for example) we decide to present this as an option.

4.11 Zero–Knowledge Proofs

In many applications, see below, we are required to provide a proof of valid encryption of an FHE ciphertext. This not only allows the encryptor to prove that they used the correct noise distribution (which avoids various selective failure attacks due to invalid noise values in higher level protocols), or at least used a distribution which is not too far from the correct one. The use of zero–knowledge proofs also allows the prover to prove that they know the underlying message (which is useful in simulation proofs where the simulator, or proof of the simulator, needs to extract any underlying messages).

In this document we provide three forms of such zero–knowledge proofs for FHE ciphertexts.
1. The first are pre–quantum secure, and are based on pairings on elliptic–curve groups; they provide very short proofs but the time to generate a proof can be long. The proofs are exact, in that they exhibit no so–called *soundness slack* (see below).
2. The second are also pre–quantum secure, however we now allow a mild form of soundness slack. In particular the prover is not able to prove that they actually used the exactly correct error distributions in forming a public key encryption. This slack is relatively mild, and will need to be coped with when setting FHE parameters. This disadvantage is compensated by the fact that these second type of proofs are more efficient than the first.
3. The third are fully post–quantum, and are based on MPC–in–the–Head techniques; they provide longer proofs than the first two, but exhibit a shorter prover time. In addition they exhibit no

soundness slack.

For the first two types of proofs, we note that a quantum computer is able to break the soundness of these first proofs, but not the zero–knowledge property. Thus privacy of the inputs is maintained in the presence of a quantum computer, but a quantum adversary will be able to prove invalid statements.

The first two (pre–quantum) proofs are also more suitable for TFHE style FHE, as they require pairing friendly elliptic curves whose group order is larger than the ciphertext modulus, whereas the third (post–quantum) proofs can be applied to any form of FHE scheme. The first two require a Common Reference String (CRS) to be securely set up via a ceremony, whilst the third does not. We summarize the properties in the Table 3.

Proof	Security	Soundness Slack	Applicability	CRS Required
Proof 1	Pre–Quantum	None	TFHE only	Yes
Proof 2	Pre–Quantum	Yes	TFHE only	Yes
Proof 3	Post–Quantum	None	All	No

Table 3: Properties of the zero–knowledge proofs.

We explain the basic idea behind our proofs, and the potential soundness slack, using TFHE as an example; adapting the methodology to BGV and BFV is easy but involves a introducing even more soundness–slack, as one needs to embed the message space modulo P into a space of bits. See Section 7.6 for further details.

4.11.1 Soundness Slack

For TFHE, as we utilize the encryption method of [Joy24] with tweaked uniform noise distributions $TUniform(1, -2^b, 2^b)$, the encryption equation can be expressed as follows (using the notation of Section 5.6.3), to produce an LWE ciphertext of dimension $\hat{\ell}$. To encrypt $m \in \mathbb{Z}/(P)$ one selects a vector $\mathbf{r} \leftarrow \{0, 1\}^{\hat{\ell}}$, a vector $\mathbf{e}_1 \leftarrow TUniform(\hat{\ell}, -2^{b_\ell}, 2^{b_\ell})$ and a value $e_2 \leftarrow TUniform(1, -2^{b_\ell}, 2^{b_\ell})$ and then one defines

$$\mathbf{a} \leftarrow \mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1,$$
$$b \leftarrow \mathfrak{pk}_b \cdot \mathbf{r} + e_2 + (Q/P) \cdot m.$$

Thus an honest prover will be providing values from the language

$$\mathcal{L} = \Big\{ (\mathbf{a}, b) : \mathbf{a} = \mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1, \ b = \mathfrak{pk}_b \cdot \mathbf{r} + e_2 + (Q/P) \cdot m,$$
$$m \in \mathbb{Z}/(P), \ \mathbf{r} \in \{0, 1\}^{\hat{\ell}}, \ \|(\mathbf{e}_1|e_2)\|_\infty \leq 2^{b_\ell} \Big\}.$$

A zero–knowledge proof will be said to have no soundness–slack if the statement that a dishonest prover can pass the soundness check is exactly the statement $\mathcal{L}$. However, our second type of pre–quantum zero–knowledge proof suffers from the issue of soundness slack. These second type of proofs are more efficient than the first type, but they allow the adversary to pass as valid a statement from the language

$$\mathcal{L}' = \Big\{ (\mathbf{a}, b) : \mathbf{a} = \mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1, \ b = \mathfrak{pk}_b \cdot \mathbf{r} + e_2 + (Q/P) \cdot m,$$
$$m \in \mathbb{Z}/(P), \ \mathbf{r} \in \{0, 1\}^{\hat{\ell}}, \ \|(\mathbf{e}_1|e_2)\|_2 \leq B \Big\},$$

for some value of B. Using the standard norm inequalities, for vectors of size n,

$$\| \cdot \|_\infty \leq \| \cdot \|_2 \leq \sqrt{n} \cdot \| \cdot \|_\infty,$$

implies we have need to utilize the language $\mathcal{L}'$ with $B = \sqrt{\hat{\ell}+1} \cdot 2^{b_{\hat{\ell}}}$. Thus we are proving statements which the adversary could select to be from the language

$$\hat{\mathcal{L}} = \Big\{ \ (\mathbf{a}, b) \ : \mathbf{a} = \mathfrak{pf}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1, \ \ b = \mathfrak{pf}_b \cdot \mathbf{r} + e_2 + (Q/P) \cdot m,$$

$$m \in \mathbb{Z}/(P), \ \ \mathbf{r} \in \{0,1\}^{\hat{\ell}}, \ \|(\mathbf{e}_1|e_2)\|_\infty \leq \sqrt{\hat{\ell}+1} \cdot 2^{b_{\hat{\ell}}} \ \Big\}.$$

This factor difference $zk\text{-}slack = \sqrt{\hat{\ell}+1}$ will be referred to as the zero–knowledge soundness slack from now on. It is the difference in bounds that the proofs guarantee between a dishonest and an honest encryptor/prover[7].

4.11.2   The Zero–Knowledge Proofs Considered as Subset–Sum Problems

Our first and third type of zero–knowledge proof all have $zk\text{-}slack = 1$, and achieve this by mapping the input language $\mathcal{L}$ into a subset–sum problem. This is done, for TFHE, as follows, since P is a power of two we can write the message as, for $m_i \in \{0,1\}$,

$$m = \sum_{i=0}^{\log_2 P} m_i \cdot 2^i.$$

For BGV and BFV we do not have such a nice bit–decomposition as P is odd, thus for BGV and BFV we need to introduce some soundness slack in terms of the underlying language in relation to the message (as opposed to the noise distribution, which was the issue above).

Again focusing on TFHE, we can rewrite the above encryption equations into a statement about unknown bits as follows:

$$\mathbf{a} = \mathfrak{pf}_a \odot \overleftrightarrow{\mathbf{r}} + E_1 \cdot \mathbf{g} - 2^{b_{\hat{\ell}}} \cdot \mathbf{1} \quad (\bmod \ Q).$$

$$b = \mathfrak{pf}_b \cdot \mathbf{r} + \left(\mathbf{e}_2 \cdot \mathbf{g} - 2^{b_{pk}} \right) + \left(\sum_{i=0}^{\log_2 P} m_i \cdot (\Delta \cdot 2^i) \right) \quad (\bmod \ Q),$$

where $\hat{\ell}$ is the ring dimension, $\odot$ is ring multiplication, $\overleftrightarrow{\mathbf{r}}$ is the reverse ordering of the vector $\mathbf{r}$, $\mathbf{g} = (1, 2, 4, \ldots, 2^{b_{\hat{\ell}}}, 1)^\top$ and $\Delta = Q/P$. The (other) unknown variables, apart from the bits making up the message m, are the bit vector $\mathbf{r} \in \{0,1\}^{\hat{\ell}}$, the bit matrix $E_1 \in \{0,1\}^{\hat{\ell} \times (b_{\hat{\ell}}+2)}$, and the bit vector $\mathbf{e}_2 \in \{0,1\}^{b_{\hat{\ell}}+2}$. By expanding out the $\odot$ operation in the above equation we can view this as $\hat{\ell}+1$ subset–sum equations. The prover is thus proving knowledge of the bits making up m, $\mathbf{r}$, E_1 and $\mathbf{e}$; thus they are proving knowledge of a simulataneous solution to a set of $\hat{\ell}+1$ subset–sum equations.

The total number of witness bits is given by

$$\log_2 P + \hat{\ell} + \hat{\ell} \cdot (b_{\hat{\ell}} + 2) + (b_{\hat{\ell}} + 2) = \log_2 P + \hat{\ell} \cdot (b_{\hat{\ell}} + 3) + (b_{\hat{\ell}} + 2).$$

The statement size on the other hand is

$$(3 \cdot \hat{\ell} + 1) \cdot \log_2 q,$$

i.e. $2 \cdot \hat{\ell}$ elements for the public key $(\mathfrak{pf}_a, \mathfrak{pf}_b)$; $\hat{\ell}$ for $\mathbf{a}$ and one for b.

4.11.3   ZKPoKs Based on Vector Commitments

Using vector commitments, it is possible to obtain very short NIZK proofs for subset–sum statements as well as short statements from the language $\mathcal{L}'$ above.

---

[7]In extensions to TFHE which we do not discuss in this document one can amortize many public key encryptions with a single proof. With this optimization the $zk\text{-}slack$ becomes $\sqrt{2 \cdot \hat{\ell}}$.

Our first type of proof based on vector commitments uses the construction of [Lib24], and relies on a vector commitment scheme [LY10]. This proof allows one to prove that a committed vector is small (i.e., binary, ternary, or of small infinity norm).

Our second type of proof based on vector commitments also uses the commitment scheme of [LY10] and its short proofs of binarity [Lib24]. In order to reduce the dimension of committed vectors and the number of exponentiations, it applies a technique from [LNS21] that consists in projecting noise vectors to smaller dimensions before proving a loose bound on their infinity norm. While this bound does not provide tight smallness guarantees by itself, it suffices to prove that a modular inner product of the noise vector with itself is also its inner product over the integers (i.e., no implicit modular reduction happens) when it comes to proving a bound on its Euclidean norm. This combination of techniques allows proving an exact bound on the Euclidean norm of a committed noise vector and a slightly slacky bound on its infinity norm. In order to prove the smallness of the Euclidean norm, we can then use an inner product argument as in [GHL22]. Instead of using BulletProofs [BBB+18] as in [GHL22], Type-2 vector-commitment-based proofs rely on the inner product functional commitment of [LRY16] which provides a constant number of group elements per proof. In order to avoid leaking the exact Euclidean norm of the noise vector and maintain the zero-knowledge property, Type-2 proofs proceed as in [GHL22] by computing the decomposition of a positive integer (namely, the difference between the squared Euclidean norm and the square of its proven upper bound) as a sum of four squares. This technique is a standard trick in range proofs and we use it in the same way as in [GHL22].

In both cases our NIZK construction relies on discrete-logarithm-hard groups $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$ of prime order r, over which a bilinear map $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ is efficiently computable. The proof requires that r is larger than the ciphertex modulus q, and thus these proofs are better suited for TFHE style FHE, for which the ciphertext modulus q is small.

The soundness property of the NIZK construction relies on the hardness of a parameterized variant of the standard discrete logarithm problem, called the (m, n)-Discrete Logarithm assumption.

Definition 1. Let $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$ be pairing-friendly groups of prime order r. For integers m, n, the (m, n)-**Discrete Logarithm** $((m, n)$-DLOG) problem is, given $(g, g^\alpha, g^{(\alpha^2)}, \ldots, g^{(\alpha^m)}, \hat{g}, \hat{g}^\alpha, \ldots, \hat{g}^{(\alpha^n)})$, for a random $\alpha \leftarrow \mathbb{Z}/(r), g \leftarrow \mathbb{G}, \hat{g} \leftarrow \hat{\mathbb{G}}$, to compute α.

The (m, n)-Discrete Logarithm assumption is used to prove the soundness of the scheme in the random oracle model and in the algebraic group model [FKL18]. This is an idealized model, where all algorithms are assumed to be algebraic. Algebraic algorithms [BV98, PV05] generalize the notion of generic algorithms [Sho97] in that, whenever they compute a group element, they do it by taking linear combinations of available group elements so far. Hence, when they output a group element $X \in \mathbb{G}$, they also output a representation $\{\alpha_i\}_{i=1}^N$ of $X = \prod_{i=1}^N g_i^{\alpha_i}$ as a function of previously observed group elements $(g_1, \ldots, g_N) \in \mathbb{G}^N$ in the same group. In contrast with generic algorithms, algebraic algorithms are given access to specific encodings of group elements. Of course, the (m, n)-Discrete Logarithm problem does not resist quantum algorithms. A quantum adversary would actually be able to generate proofs for false statements and break the soundness of the proof system. However, the zero-knowledge property is perfect (i.e., a simulator can use α as a trapdoor to simulate proofs that have exactly the same distribution as real proofs) and does not rely on any assumption. Consequently, if the proof system is used to prove the validity of an LWE/RLWE ciphertext, the post-quantum security of the encryption scheme is preserved since proofs leak no information on the witnesses.

The first vector-commitment-based construction provides the same proof length as size-optimal pairing-based SNARKs [Gro16]. The second construction requires longer proofs (namely 13 group elements, which fit in about 1KB on BLS12-446 curves) but are much faster to compute. For typical parameters, the dimension of committed vectors (which linearly affects the number of group exponentiations at the prover) is reduced by a factor ≈ 9. Technically, the vector-commitment-based proofs are not SNARKs since the verification time grows with the length of the witness. Even in the fast-verification

variant (where the verifier computes only ≈ 128 group exponentiations), the verifier still has to compute a number of field operations that linearly grows with the witness length. However, the prover is significantly faster than in pairing-based SNARKs featuring similar proof sizes for general NP statements. As discussed in [Lib24, Appendix G.5], for the specific task of proving the validity of a ciphertext in Joye's public-key scheme [Joy24], the prover would have to prove an arithmetic circuit with about 90000 multiplication gates and 280000 wires, which would cost over 500000 exponentiations. In Type-2 vector-commitment-based proofs, the prover only computes around 60000 exponentiations in pairing-friendly groups.

As another advantage over SNARKs, vector-commitment-based proofs extend to prove other statements[8] about encrypted messages without any modification in the CRS. In non-universal SNARKs like [Gro16], this would require to generate a new CRS for each different circuit. For universal SNARKs with pre-processing (e.g., [GWC19]), it would require a new circuit-dependent pre-processing phase.

4.11.4   ZKPoKs Based on MPC-in-the-Head

As mentioned above we also present ZKPoKs which are post-quantum secure, and based on the MPC-in-the-Head paradigm. These provide proofs sizes of the order of a few thousand kilobytes, thus much larger than those based on vector commitments; however the prover time is relatively fast.

Our ZKPoK will be based on the KKW MPC-in-the-Head protocols from [FMRV22] (for general values of Q) and [FR23] (for the case of prime values of $Q = p_1 \cdots p_k$). We also present an optimized version of the [FMRV22] ZKPoKs for subset-sum problems based on a nice hyper-cube trick. All these techniques are themselves based on the classic KKW [KKW18] approach to MPC-in-the-Head. Since our encryption scheme is essentially a linear function of random bits, our MPC-in-the-Head protocol does not need to actually execute any multiplication gates. We thus adopt the cut-and-choose methodology to check the offline phase of the MPC-in-the-Head protocol, as opposed to the sacrificing method of [HKL21]; this is because the checks using sacrificing would require multiplications and hence be more costly in the long run. Thus our security relies on the combinatorial arguments of the original KKW paper, despite our "arithmetic" circuit being over a large ring $\mathbb{Z}/(Q)$ and not simply a binary circuit.

The core idea underlying MPC-in-the-Head proof systems is for the prover to emulate an MPC protocol that computes the verification circuit of the decision problem at hand, where the secret that is shared amongst the virtual parties is the witness to the statement. By emulating the MPC protocol, the prover can commit to the view of every party towards the verifier who can in turn challenge to open some of these views. By the correctness property of the MPC computation, examining enough parties' views and checking that these views are coherent with one another, convinces the verifier of the integrity of the computation and thus of the soundness of the proof. In addition, the privacy property of the MPC computation ensures that by keeping enough of the parties' views secret, the verifier is not able to reconstruct any of the secrets, providing the zero-knowledge property.

The security of the proof therefore only relies on information theoretic arguments and the ability to compute commitments, for which we know post-quantum constructions. As such, signature schemes that internally rely on an MPC-in-the-Head proof system to build an identification scheme have been proposed as candidates for post-quantum signature schemes [CDG+17, BBD+23]. We can turn this interactive process into a NIZK via the use of the Fiat-Shamir transform at the cost of being able to claim security only in the Random Oracle model.

In contrast to SNARKs or ZKPoKs based on vector commitments, proofs generated using the MPC-in-the-Head paradigm are not short nor are they fast to verify, but they can be faster to create the proof. In fact, the size, as well as the proving and verification time of an MPC-in-the-Head proof, are all linear in both the number of secret inputs and the number of multiplications in the circuit. Another advantage

---

[8]As explained in [Lib24, Appendix G.7], they make it possible to prove plaintext equalities, inequalities < between encrypted integers, or that two plaintexts agree in certain bit positions. It is also possible to prove that at most one-out-of-k ciphertexts encrypts a non-zero value. For a given ciphertext, it can also prove an upper bound on the Hamming weight of the plaintext.

of MPC–in–the–Head proofs is that despite their linear–time nature, both the proving and verification process are highly parallelizable as each party's view can be computed concurrently.

4.12 Putting it all Together

The reason for presenting both threshold FHE protocols and ZKPoKs of correct ciphertext encryption is that together they enable secure, robust low communication MPC–like protocols based on FHE.

The traditional client–server application of FHE is that a client generates an FHE key, and sends the public part (for evaluation) to the server. Then the client can send encrypted data to the server, and the server can compute on it. The encrypted results are then passed back to the client for decryption. In this application one assumes the server is semi–honest, and has no secret inputs. Thus the client is the honest party, and so obtains no advantage by crafting invalid ciphertexts.

The classic usage of threshold decryption would be to enable other parties, and not just a single client to decrypt data. In such a situation it is highly likely that other parties may be encrypting data as well. For example consider a financial application in which customers encrypt their transaction data, the financial company processes the transactions in encrypted form, and then each customer may request a decryption of their balance from a network of parties implementing the threshold decryption protocols above. In such a situation customers, i.e. encryptors, cannot be assumed to be honest, and so they need to ensure that encrypted data entered into the system is via valid ciphertexts.

Indeed, one can consider the application of FHE in the financial sector of the previous paragraph as an example MPC application; where the parties are the customer, the financial institution and the parties enabling the threshold decryption to be performed. One sees that FHE enables a round efficient MPC protocol relatively simply. Indeed, if one utilizes a multitude of parties to perform the function evaluation (for example multiple banks in the above financial use case), one can obtain a fully malicious MPC protocol with low round complexity; and whose communication complexity does not depend on the complexity of the functions being evaluated.

That the FHE–enabled MPC protocol above is fully malicious is due to the combination of the robust threshold decryption protocols and the application of ZKPoKs for correct encryption given in this document. A complete security model, and security proof, of such an FHE–enabled MPC protocol is given in [Sma23]. This FHE–enables MPC protocol utilizes the blueprint of Gentry's passively secure MPC protocol (given in [Gen09], and obtains active security by combining the basic FHE operations with ZKPoKs of correct encryption, and a robustly secure threshold key generation and encryption method for the FHE primitive.

5   Conventional Primitives and Schemes

The purpose of this whole document is to specify **threshold** Fully Homomorphic Encryption; it is not a document which aims to describe a potential FHE standard. Thus, following Design Decision 16, Design Decision 18 and Design Decision 19, we only present enough FHE material in order to implement the threshold schemes, see the subtleties needed to implement the threshold FHE schemes, and also provide a **simple** FHE implementation. There are a huge number of optimizations and bells–and–whistles one can apply in the FHE space, and we do not aim to cover them here. Our main focus is on building threshold TFHE, thus we only present BGV and BFV (and their threshold variants) in order to show that our techniques are general in nature. Some of our threshold techniques are not so well suited to BGV and BFV, especially in the regime where one has a large number of parties.

5.1   Extendable Output Functions

A standard primitive from prior work which we shall also use in our work is a so–called Extendable Output Function (XOF). A XOF is a keyed pseudo–random function (PRF), which can produce arbitrary length outputs, see [MF21][Chapter 13.3]. The standard XOF construction is to use SHAKE–256, based on the SHA–3 sponge to squeeze. This is defined as follows:

$$SHAKE\text{-}256(M, d) = Keccak[512](M\|1111, d)$$

where $Keccak[512](m, d)$ is the Keccak operation with capacity $c = 512$ (and hence rate $r = 1088$) applied to the message m, and outputting d bits. The $Keccak$ operation produces r bits of output in each iteration of the squeeze. We write the API for this application of $Keccak$ as $Keccak[512].Init(m)$ and $Keccak[512].Next()$, where the latter outputs the next r bits of output.

    Whilst theoretically nice, the $SHAKE\text{-}256$ based XOF is relatively slow, thus as a less theoretically pleasing method, one can use an AES–CTR based approach. This approach does not produce a XOF, but is sufficient for the applications we envisage. This method works much like CTR–DRBG, however as the output of our XOF applications is always *public* we do not need to worry about the frequent rekeying used in the CTR–DRBG standard. Thus our AES–based "XOF" is simpler in construction than CTR–DRBG. The construction is to compress the input message m down to a two block output using a fixed key AES CBC–MAC. This produces a key and an IV. The key and the IV are then used in AES–CTR mode to produce an output stream 128–bits at a time.

    Whether using $SHAKE\text{-}256$ or an AES–based XOF, we define the base XOF object in Figure 3. It has two functions $XOF.Init(seed, str)$ which initializes the basic XOF object from a seed value $seed$ and a domain separation string str, and a function $XOF.Next(n)$ which outputs the next n bits of output. Note, the domain separation string str is assumed to be bounded in length by at most five characters in this document.

    To aid implementation in software our XOF objects internally process data in bytes rather than bits; see $XOF.Next(n)$ in Figure 3. This means if, for example, nine bits are requested from the XOF, the algorithm internally consumes two bytes of the output, throws away seven bits, and then returns nine bits. Alternatively, an implementation in such a situation may decide to return the full two bytes (i.e. sixteen bits) and the application only utilizes the desired nine.

    In our application of a XOF we may want the output not to be bits, but to be a vector of N uniformly random elements from $(\mathbb{Z}/(Q))^N$. When Q is a power of two we trivially obtain this, but when Q is not a power of two we use a method (see $XOF.Next(N, Q)$ in Figure 3) to obtain a distribution which is within 2^{-dist} of uniform.

```
┌─ XOF ─────────────────────────────────────────────────────────────────────┐
│                                                                            │
│  XOF.Init(seed, str):                                                      │
│      1. If using SHAKE-256 then                                            │
│          (a) Keccak[512].Init(seed‖str‖1111).                              │
│      2. Else                                                               │
│          (a) Pad seed‖str with zeros to the right, in order to obtain a message m = m_1, …, m_t │
│              of t > 2 128–bit blocks, with the last block being the encoding of the length of │
│              seed‖str in bits.                                             │
│          (b) k ← 0, c_0 ← 0.                                               │
│          (c) For i = 1, …, t do c_i ← AES_k(c_{i-1} ⊕ m_i).               │
│          (d) IV_XOF ← c_{t-1}.                                            │
│          (e) k_XOF ← c_t.                                                  │
│      3. buff_XOF = ∅.                                                      │
│                                                                            │
│                                                                            │
│  XOF.Next(n):                                                              │
│      1. While |buff_XOF| < 8 · ⌈n/8⌉ do                                   │
│      2. If using SHAKE-256                                                 │
│          (a) buff_XOF ← buff_XOF‖Keccak[512].Next()                        │
│      3. Else                                                               │
│          (a) buff_XOF ← buff_XOF‖AES_{k_XOF}(IV_XOF).                      │
│          (b) IV_XOF ← IV_XOF + 1.                                          │
│      4. Write r‖buff ← buff_XOF, where r is ⌈n/8⌉ bytes in length.        │
│      5. buff_XOF ← buff.                                                   │
│      6. Return the first n bits of r, or the whole of r as desired.        │
│                                                                            │
│                                                                            │
│  XOF.Next(N, Q):                                                           │
│      1. For i = 1, …, N do                                                 │
│          (a) If Q is a power of two s ← XOF.Next(log_2 Q).                 │
│          (b) Else s ← XOF.Next(dist + ⌈log_2 Q⌉).                         │
│          (c) Treating the bit–string s as an integer x_i ← s (mod Q).     │
│      2. Output (x_1, …, x_N).                                              │
│                                                                            │
└────────────────────────────────────────────────────────────────────────────┘
```

Figure 3: Definition of the XOF Object.

5.2 (Generalized) (Ring) Learning with Errors

Our FHE ciphertexts will come with plaintext modulus P, and ciphertext modulus $Q = Q_1 \cdots Q_L$, where the factors in the ciphertext modulus are co–prime, i.e. $\gcd(Q_i, Q_j) = 1$ for $i \neq j$, but each Q_i could be a prime power (or composite number).

We will often make use of the M–th cyclotomic ring of degree $N = \phi(M)$ which we will define by

$$\mathbf{R}^{(M)} = \mathbb{Z}[X]/(\Phi_M(X)).$$

In "most" instances users will use $M = 2^{n+1}$, and so $N = 2^n$, and the ring will be given by

$$\mathbf{R}^{(M)} = \mathbb{Z}[X]/(X^N + 1).$$

From now on we will implicitly assume that we have $M = 2^{n+1}$ and $N = 2^n$ unless otherwise stated. In addition, if the cyclotomic value M is obvious from the context we will drop the superscript and simply write $\mathbf{R}$.

The reduction of the ring modulo the ciphertext (resp. plaintext) modulus Q (resp. P) will be denoted by

$$\mathbf{R}_Q = \mathbf{R}_Q^{(M)} = (\mathbb{Z}/(Q))[X]/(\Phi_M(X)) \quad (\text{resp. } \mathbf{R}_P = \mathbf{R}_P^{(M)} = (\mathbb{Z}/(P))[X]/(\Phi_M(X))).$$

We fix the global Δ as $\Delta = \lfloor Q/P \rfloor$. This is the ratio between the ciphertext modulus Q, and the *application* plaintext modulus P.

Elements in $\mathbf{R}$ (resp. $\mathbf{R}_Q$, $\mathbf{R}_P$, etc) will be considered as vectors $\mathbf{a}$, $\mathbf{b}$, etc where we apply the component-wise addition operation. However, we will let $\mathbf{a} \cdot \mathbf{b}$ denote the dot–product of two such vectors (resulting in an element of $\mathbb{Z}$), and $\mathbf{a} \odot \mathbf{b}$ to denote the ring multiplication (resulting in an element of $\mathbf{R}$). This mapping between vectors and ring elements will be via the identification of the vector $\mathbf{a} = (a_0, \ldots, a_{N-1})$ with the ring element

$$\mathbf{a} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1}.$$

We let $\overleftrightarrow{\mathbf{a}}$ denote the vector with the coefficients in the reverse order, so that if $\mathbf{a} = (a_0, \ldots, a_{N-1})$ then $\overleftrightarrow{\mathbf{a}} = (a_{N-1}, \ldots, a_0)$. An important identity, which will be used in our method for public key encryption for TFHE is that the $(N-1)$–th element in the product of the two ring elements $\mathbf{a} \odot \overleftrightarrow{\mathbf{b}}$ is equal to the normal dot product of the two vectors $\mathbf{a} \cdot \mathbf{b}$. Thus we have

$$\mathbf{a} \cdot \mathbf{b} = (\mathbf{a} \odot \overleftrightarrow{\mathbf{b}})_{N-1}.$$

We utilize a number of variants of the Learning–with–Errors problem in this document. The (standard) LWE problem, the Ring–LWE problem, and the Generalized Ring–LWE problem. We first define these problems

Definition 2 (Learning–With–Errors). The (decision) standard LWE problem is to distinguish between samples drawn from the two distributions

$$D_1 = \{ (\mathbf{a}, b) \ : \ \mathbf{a} \leftarrow (\mathbb{Z}/(Q))^{\ell}, \ b \leftarrow \mathbb{Z}/(Q) \},$$
$$D_2 = \{ (\mathbf{a}, b) \ : \ \mathbf{a} \leftarrow (\mathbb{Z}/(Q))^{\ell}, \ e \leftarrow \mathcal{D}, \ b = \mathbf{a} \cdot \mathbf{s} + e \},$$

where $\mathbf{s} \in (\mathbb{Z}/(Q))^{\ell}$ is a fixed (secret) value, and $\mathcal{D}$ is an LWE–error distribution producing "small" elements in $\mathbb{Z}/(Q)$.

Definition 3 (Ring Learning–With–Errors). The (decision) Ring–LWE problem is to distinguish between samples drawn from the two distributions

$$D_1 = \{ (\mathbf{a}, \mathbf{b}) \ : \ \mathbf{a}, \ \mathbf{b} \leftarrow \mathbf{R}_Q \},$$
$$D_2 = \{ (\mathbf{a}, \mathbf{b}) \ : \ \mathbf{a} \leftarrow \mathbf{R}_Q, \ \mathbf{e} \leftarrow \mathcal{D}, \ \mathbf{b} = \mathbf{a} \odot \mathbf{s} + \mathbf{e} \},$$

where $\mathbf{s} \in \mathbf{R}_Q$ is a fixed (secret) value, and $\mathcal{D}$ is an LWE–error distribution producing "small" elements in $\mathbf{R}_Q$.

Definition 4 (Generalized–Ring–Learning–With–Errors). The (decision) Generalized–Ring–LWE problem is to distinguish between samples drawn from the two distributions

$$D_1 = \{ (\mathbf{a}_0, \ldots, \mathbf{a}_{w-1}, \mathbf{b}) \ : \ \mathbf{a}_i, \mathbf{b} \leftarrow \mathbf{R}_Q \},$$
$$D_2 = \{ (\mathbf{a}_0, \ldots, \mathbf{a}_{w-1}, \mathbf{b}) \ : \ \mathbf{a}_i \leftarrow \mathbf{R}_Q, \ \mathbf{e} \leftarrow \mathcal{D}, \ \mathbf{b} = \sum_i \mathbf{a}_i \odot \mathbf{s}_i + \mathbf{e} \},$$

where $\mathbf{s}_0, \ldots, \mathbf{s}_{w-1} \in \mathbf{R}_Q$ are fixed (secret) values, and $\mathcal{D}$ is an LWE–error distribution producing "small" elements in $\mathbf{R}_Q$.

The Generalized–Ring–LWE problem was first introduced in [BGV12] with the name GLWE, it is some-times in more modern literature referred to as the Module–LWE problem [LS15]. Clearly the standard Ring–LWE is simply the Generalized–Ring–LWE problem where we set $w = 1$. From a cryptanalysis point of view the best attacks on all three problems are simply to ignore the generalized/ring nature of the problems and just to attack the underlying standard LWE problem of dimension ℓ (resp. N and $w \cdot N$). To

obtain suitable security parameters it is common to utilized the **lwe-estimator** [APS15], which we shall do in this work.

As nice rules of thumb:

- For fixed $w \cdot N$ and Q, as we increase the standard deviation σ of the coefficients in the "noise" term **e**, the security increases.
- For fixed $w \cdot N$ and σ, as we decrease the value of Q, the security increases.

The first rule of thumb means we only need consider security of fresh ciphertexts, as performing homomorphic operations results in increased values of the standard deviation of the noise term. The second rule of thumb also implies that reducing the ciphertext modulus whilst maintaining the size of the noise, can only increase security; so again we only need consider security of fresh (or freshly bootstrapped) ciphertexts at the top most level, in the case of BGV/BFV style systems.

It is traditional to model $\mathcal{D}$ as a distribution which outputs vectors of size ℓ (resp. polynomials of degree less than N) whose elements (resp. coefficients) come from a discrete version of the Gaussian distribution with "small" standard deviation σ. For appropriate values of the parameters (N, w, Q, σ) the problems are believed to be hard. It is conjectured (and we will assume) that the decision LWE problem is still hard when one makes the following alterations (all of which are standard modifications).

1. **s** is chosen from $\{0, 1\}^N$, or $\{-1, 0, 1\}^N$ (depending on the scheme).
   - In the former case we select zero and one with equal probability 1/2.
   - In the latter case we select zero with probability 1/2, and one/minus one each with probability 1/4. Note, this outputs entries with standard deviation $\sigma = \sqrt{1/2}$. In this document we shall call this distribution on $\{-1, 0, 1\}^N$ by the name $NewHope(N, 1)$.

   The distribution $\{0, 1\}^N$ is usually used for the secret key distribution for TFHE, whilst the distribution $NewHope(N, 1)$ is used for the secret key distribution for schemes such as BGV and BFV.

2. In the case of TFHE, the noise values e are chosen from the tweaked uniform distribution, on $[-2^b, \ldots, 2^b]$, introduced earlier as $TUniform(1, -2^b, 2^b)$, with $\sigma^2 = (2^{2 \cdot b + 1} + 1)/6$, i.e. we replace the usual discrete Gaussian with standard deviation σ' by a distribution with mean zero and with standard deviation $\sigma \geq \sigma'$.

3. In the case of BGV/BFV, the noise values **e** are chosen from the set of polynomials whose vector of coefficients is chosen via the distribution $NewHope(N, B)$. i.e. we replace the usual discrete Gaussian with standard deviation σ' by a distribution with standard deviation $\sigma_B = \sqrt{B/2} \geq \sigma'$.

For the TFHE parameters we have smaller values of Q and N, and hence we need larger standard deviations in the noise. For the BGV and BFV parameters the standard deviation in the noise term can be very small indeed, as Q and N are much bigger. This dictates the differences in noise distributions used in both schemes.

It is (relatively) hard within an MPC engine to sample approximations to Discrete Gaussians with large standard deviation, whereas sampling from $TUniform(N, -2^b, 2^b)$ and $NewHope(N, B)$ are simpler within an MPC engine. In addition, such distributions can provide exact zero-knowledge proofs (i.e. with no soundness slack) of correct encryption, i.e. it is also easier to utilize distributions which are easier to sample in MPC systems with the zero-knowledge proofs.

To aid parameter generation we provide the tables in Appendix A. These list for the two different secret key distributions and different sizes of Q, and b, the minimum value of $w \cdot N$ (which we select to be a multiple of 256 for the TFHE case, and a multiple of 5000 for the BGV/BFV case), for which the lattice estimator [APS15] provides an LWE security estimate which is larger than 130, we also list (for interest) the actual security value that the LWE estimator provides for this value of N. This gives an estimate of how much security margin we have for these parameters.

For the purposes of reproducibility the security estimates were obtained using the LWE-estimator calls:

```
if TFHE:
```

```
    params = LWE.Parameters(wN, Q, Xs = ND.Uniform(0,1), Xe = ND.TUniform(B))
  else:
    params = LWE.Parameters(wN, Q, Xs = ND.CenteredBinomial(1),
                                   Xe = ND.CenteredBinomial(B))
  if wN>10000:
    est = LWE.estimate(params, deny_list = ("arora-gb", "bkw", "bdd_hybrid",
                                            "bdd_mitm_hybrid", "dual_mitm_hybrid"))
  else:
    est = LWE.estimate(params, deny_list = ("arora-gb", "bkw"))
```

We avoid calling the *arora–gb* and *bkw* methods, as these are slow arnd are hardly ever the fastest attack methods. For large dimensions we also avoid calling any hybrid methods, as they are numerical unstable for very large values of Q, are slow, and again (for large dimension) unlikely to lead to an improvement on other methods.

5.3 Ring Multiplication

The main computation expense in all ring–LWE based schemes is the evaluation of the product $\mathbf{a} \odot \mathbf{b}$ for two ring elements $\mathbf{a}$ and $\mathbf{b}$ represented as polynomials;

$$\mathbf{a} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1},$$
$$\mathbf{b} = b_0 + b_1 \cdot X + \cdots + b_{N-1} \cdot X^{N-1}.$$

As N is a power of two, and we are using the cyclotomic polynomial $X^N + 1$ to define the ring $\mathbf{R}_Q$, the naive method to compute this product is to evaluate

$$\mathbf{a} \odot \mathbf{b} = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-i-1} a_i \cdot b_j \cdot X^{i+j} - \sum_{j=N-i}^{N-1} a_i \cdot b_j \cdot X^{i+j-N} \right) \pmod{Q}.$$

This operation requires $O(N^2)$ underlying operations modulo Q. Using Karatsuba style techniques one could reduce this to $O(N^{\log_2(3)})$. This can be reduced to $O(N \cdot \log N)$ by use of either the FFT or NTT algorithms.

5.3.1 FFT Based Multiplication

When Q is relatively small (in the case of TFHE), one can perform the multiplication of the polynomials by using the standard complex FFT algorithm. We take θ to denote a complex primitive $(2 \cdot N)$–th root of unity, and then implement the algorithm as in Figure 4.

FFT–Based $\mathbf{a} \odot \mathbf{b}$

$\mathbf{c} + \mathbf{e} \leftarrow \mathbf{a} \odot \mathbf{b}$:
  Where $\mathbf{e}$ is "small". In what follows $\theta \in \mathbb{C}$ is a primitive $(2 \cdot N)$–th root of unity.
  1. $\overline{\mathbf{a}} \leftarrow FFT(\mathbf{a}, \theta) \in \mathbb{C}^N$.
  2. $\overline{\mathbf{b}} \leftarrow FFT(\mathbf{b}, \theta) \in \mathbb{C}^N$.
  3. For $i \in [0, \ldots, N-1]$ do $\overline{c}_i \leftarrow \overline{a}_i \cdot \overline{b}_i$.
  4. $\mathbf{t} \leftarrow FFT^{-1}(\overline{\mathbf{c}}, \theta)$.
  5. Round $\mathbf{t}$ to an integer vector and reduce it mod Q.
  6. Output $\mathbf{t} = \mathbf{c} + \mathbf{e}$.

Figure 4: Non–Exact FFT–Based Ring Multiplication.

However, this use of the FFT comes with some drawbacks. To perform the FFT we utilize complex numbers which can only be approximated within a computer, and not held exactly. Of course, if one was willing to use very high precision real numbers then one could perform an exact evaluation of the algorithm in Figure 4. But such use of large precision real numbers would provide a performance bottleneck.

Thus in practice one approximates the real numbers in the FFT (and associated inverse–FFT) operations by standard IEEE floating point double precision (in the case of $Q = 2^{64}$), and a modification to higher precision (in the case of $Q = 2^{128}$). We call these two representations $float64$ and $float128$, they are defined (in the (almost) usual way[9]) by

  · $float64$: Mantissa of 53 bits, exponent of 11 bits, and one sign bit.
  · $float128$: Mantissa of 106 bits, exponent of 22 bits, and one sign bit.

This means that the algorithm in Figure 4 is never evaluated exactly, but is evaluated in an approximate manner. We say the algorithm is non–exact, and it implicitly generates an error vector **e**. This extra error vector **e** will turn out not to be a problem in practice, as the "noise" it creates can be absorbed into the noise terms in the underlying LWE ciphertexts; we shall return to this point below.

5.3.2 NTT Based Multiplication

When Q is large (in the case of BGV and BFV) one can select $Q = Q_1 \cdots Q_L$ so that all the divisors Q_i are prime, and satisfy $Q_i \equiv 1 \pmod{2 \cdot N}$. This means, via the Chinese Remainder Theorem, one can find a value $\theta \in \mathbb{Z}/(Q)$ which is a simultaneous primitive $(2 \cdot N)$–th root of unity for all primes Q_i dividing Q. The exact same algorithm for evaluating the FFT can then be used in the mod–Q domain, where it is now called the NTT. This results in a multiplication algorithm given in Figure 5. The output of this multiplication is however exact, unlike the case of using the FFT algorithm. In order to use this optimization, leads us to the design decision:

> **Design Decision 24: Restrictions on Q for NTT Based Multiplication**
>
> We assume, for BGV and BFV, that the ciphertext modulus Q is a product of primes $Q = Q_1 \cdots Q_L$, where the primes Q_i and the extra large prime R for key switching are chosen to satisfy $Q_i \equiv R \equiv 1 \pmod{2 \cdot N}$ for all i.

> **NTT–Based $\mathbf{a} \odot \mathbf{b}$**
>
> $\mathbf{c} \leftarrow \mathbf{a} \odot \mathbf{b}$:
>   In what follows $\theta \in \mathbb{Z}/(Q)$ is a primitive $(2 \cdot N)$–th root of unity for all primes Q_i dividing Q
>   1. $\overline{\mathbf{a}} \leftarrow NTT(\mathbf{a}, \theta) \in \mathbb{Z}/(Q)^N$.
>   2. $\overline{\mathbf{b}} \leftarrow NTT(\mathbf{b}, \theta) \in \mathbb{Z}/(Q)^N$.
>   3. For $i \in [0, \ldots, N-1]$ do $\overline{c}_i \leftarrow a_i \cdot b_i \pmod{Q}$.
>   4. $\mathbf{c} + \mathbf{e} \leftarrow NTT^{-1}(\overline{\mathbf{c}}, \theta)$.

Figure 5: Exact NTT–Based Ring Multiplication.

5.3.3 The FFT/NTT Algorithms

Since in our application N is a power of two, the FFT and NTT algorithm which we utilize are relatively simple. Indeed the same algorithm is essentially used for both the FFT and the NTT algorithm. The only

---

[9]The $float128$ type is obtained (in our code) by using two $float64$ types to emulate $float128$, leading to a mantissa of 106 as opposed to following "standardized" 128–bit float arithmetic, which does not have hardware support on most processors.

difference being the domain over which the algorithm is executed, i.e. either the complex numbers $\mathbb{C}$ or the ring $\mathbb{Z}/(Q)$. The forward and inverse algorithms are given in Figure 6.

FFT/NTT Algorithms

In what follows we use *FFT* and assume $\theta \in \mathbb{C}$. When $\theta \in \mathbb{Z}/(Q)$ one should syntactically change *FFT* to *NTT* below, and leave all else unchanged.

FFT($\mathbf{a}, \theta$):
  On input $\mathbf{a}$ is a vector of length N, a power of two.
  1. If $N = 1$ then return.
  2. Let $\mathbf{b}$ and $\mathbf{c}$ denote vectors of length $N/2$.
  3. For $i \in [0, \ldots, N/2 - 1]$ do $b_i \leftarrow a_{2 \cdot i}$ and $c_i \leftarrow a_{2 \cdot i + 1}$.
  4. $\beta \leftarrow \theta^2$.
  5. $\mathbf{b} \leftarrow FFT(\mathbf{b}, \beta)$.
  6. $\mathbf{c} \leftarrow FFT(\mathbf{c}, \beta)$.
  7. $\omega \leftarrow \theta$.
  8. For $i \in [0, \ldots, N/2 - 1]$:
     (a) $\tau \leftarrow \omega \cdot c_i$.
     (b) $\overline{a}_i \leftarrow b_i + \tau$.
     (c) $\overline{a}_{i+N/2} \leftarrow b_i - \tau$.
     (d) $\omega \leftarrow \omega \cdot \beta$.
  9. Return $\overline{\mathbf{a}}$.

FFT^{-1}($\overline{\mathbf{a}}, \theta$):
  On input $\mathbf{a}$ is a vector of length N, a power of two.
  1. $\mathbf{a} \leftarrow FFT(\overline{\mathbf{a}}, \theta^{-2})$.
  2. For $i \in [0, \ldots, N - 1]$ do $a_i \leftarrow a_i/N$.
  3. Return $\mathbf{a}$.

Figure 6: The Forward and Inverse FFT/NTT Algorithms.

5.4 Norms and Noise Analysis of LWE Ciphertexts

5.4.1 Norms of Elements in $\mathbb{Z}/(Q)$

For a scalar value $x \in \mathbb{Z}/(Q)$ we let $\|x\|$ denote the absolute value of the centred reduction of x, i.e. we represent $x \in [-Q/2, \ldots, Q/2]$ and then output the absolute value of this representation.

In analyzing noise estimates for our schemes we estimate the standard deviation of values such as $x \in [-Q/2, \ldots, Q/2]$, and assume the value x is distributed close to a Gaussian with the given standard deviation. In practice x will result from various operations consisting of summing large numbers of random variables, and hence the "assumption" of Gaussian'ness is somewhat justified by the central limit theorem. This assumptions allows us to derive high probability bounds on values such as $\|x\|$ in terms of the stated standard deviations.

Suppose it is desired to obtain a given error probability *err* (here think of $err = 2^{-40}$ or 2^{-64} or 2^{-80} or 2^{-128}, but due to the IND–CPA^D attacks mentioned earlier we focus in this document on $err = 2^{-128}$). For this we use the *complimentary error function erfc*, which is defined by

$$erfc(z) = 1 - \frac{2}{\sqrt{\pi}} \cdot \int_0^z e^{-t^2} dt.$$

In particular we have the classical estimate

$$\Pr\left[\|x\| > c \cdot \sigma\right] \leq erfc(c/\sqrt{2}),$$

where σ is the standard deviation of the value $x \in [-Q/2, \ldots, Q/2]$. The constant c does not need to be that large, for example we have the values in Table 4. Thus to obtain a right hand side of the above equation to be less than $err \approx 2^{-128}$ we can set,

$$c_{err,1} = 13.15$$

as a fixed constant for c, where we use the notation $c_{err,1}$ to indicate this is for a "ring" of dimension one. See Section 4.10 as to why we use $err \approx 2^{-128}$ and Section 5.4.2 for the generalization to large rings.

c	$-\log_2(\,erfc(c/\sqrt{2})\,)$
4	13.9
5	20.7
6	28.9
7	38.5
8	49.5
9	61.9
10	75.8
11	91.1
12	107.8
13	125.9
14	145.5

Table 4: Table of values of $-\log_2(\,erfc(c/\sqrt{2})\,)$.

5.4.2 Norms of Elements in $\mathbf{R}_Q$

For $\boldsymbol{\alpha} \in \mathbf{R}_Q$ we let $\|\boldsymbol{\alpha}\|_\infty$ denote the infinity–norm of the vector $\boldsymbol{\alpha}$, with coefficients in $(-Q/2, \ldots, Q/2]$, when considering $\boldsymbol{\alpha}$ as a polynomial (we say $\boldsymbol{\alpha}$ is in the polynomial–basis). The norm $\|\boldsymbol{\alpha}\|_\infty$ on $\mathbf{R}_Q$ is the natural one for decryption purposes in BGV and BFV. But there is a more natural one for arithmetic purposes. If we let $\theta_1, \ldots, \theta_N$ denote the cyclotomic primitive roots of unity, then from $\boldsymbol{\alpha}$ we can consider the *canonical embedding* of the elements $\boldsymbol{\alpha}$ into N–copies of the complex numbers $\mathbb{C}^N$ given by

$$\boldsymbol{\alpha} \longrightarrow can(\boldsymbol{\alpha}) = (\boldsymbol{\alpha}(\theta_1), \ldots, \boldsymbol{\alpha}(\theta_N)),$$

where we consider $\boldsymbol{\alpha}$ as being a polynomial in $\mathbb{Z}[X]$ of degree bounded by N with coefficients in $(-Q/2, \ldots, Q/2]$. The canonical norm $\|\boldsymbol{\alpha}\|^{can}$ of $\boldsymbol{\alpha}$ is then the infinity norm of the vector in the canonical embedding above, i.e. $\|\boldsymbol{\alpha}\|^{can} = \|can(\boldsymbol{\alpha})\|_\infty$.

The canonical norm is especially useful due to the following Lemma.

Lemma 1. *For all $\boldsymbol{\alpha}, \boldsymbol{b} \in \mathbf{R}$ and $\lambda \in \mathbb{Z}$ we have that*
- $\|\boldsymbol{\alpha} + \boldsymbol{b}\|^{can} \le \|\boldsymbol{\alpha}\|^{can} + \|\boldsymbol{b}\|^{can}$.
- $\|\lambda \cdot \boldsymbol{\alpha}\|^{can} = \lambda \cdot \|\boldsymbol{\alpha}\|^{can}$.
- $\|\boldsymbol{\alpha} \odot \boldsymbol{b}\|^{can} \le \|\boldsymbol{\alpha}\|^{can} \cdot \|\boldsymbol{b}\|^{can}$.

In addition, there is a constant E_M (sometimes called the ring constant) such that for all $\boldsymbol{\alpha} \in \mathbf{R}_Q$ we have

$$\|\boldsymbol{\alpha}\|_\infty \le E_M \cdot \|\boldsymbol{\alpha}\|^{can}. \tag{1}$$

This allows us to pass from bounds on the canonical norm, to bounds on the standard infinity–norm. The problem is that E_M for arbitrary M can be very large. In the important case when M is a prime power we have $E_M = 1$. For more general M one may be led to consider non–polynomial bases, for example the so–called powerful–basis. This added complication can be done for methods in this document, see

[HS20] for a discussion. We will assume going forward that the ring constant E_M is sufficiently small so that we can utilize the polynomial basis.

> ### Design Decision 25: Ring–LWE Ring Constant Size
>
> We assume the ring **R** is chosen so that the ring constant E_M is small. In particular by choosing **R** to be a power–of–two cyclotomic ring we can assume that $E_M = 1$.

In analyzing probabilistic bounds on the values in our schemes we make use of the methodology described in Appendix A of the full version of [GHS12] and in [HS20]. We note the following only provides a crude analysis for the case of general M/N, which is suitable for our purposes (again, we re–iterate we are not writing here a full standard/specification of Fully Homomorphic Encryption).

Let **α** denote some random variable in $\mathbf{R}_Q$ whose coefficients (in the polynomial embedding) are selected from a symmetric distribution with standard deviation σ. That the distribution is symmetric implies that the expectation of each coefficient is zero. The value $can(\mathbf{α})$ is then a random variable consisting of N complex values, whose expectation is also zero, and whose standard deviation is $\sigma^{can} = \sigma \cdot \sqrt{N}$. For example, if **α** is chosen with coefficients chosen from the distribution $TUniform(N, -2^b, 2^b)$ then the standard deviation of $can(\mathbf{α})$ is $\sqrt{N \cdot (2^{2 \cdot b_\ell + 1} + 1)/6}$. If the vector **α** is chosen with coefficients chosen from the distribution $NewHope(N, B)$ then the standard deviation of $can(\mathbf{α})$ is $\sqrt{B \cdot N/2}$.

We can use the $erfc$ function to find an upper bound on the size of the i–th component of $can(\mathbf{α})$, i.e. $|can(\mathbf{α})_i|$ which holds with probability $1 - err$. This can be done as the law of large number can be applied since N is "large", and so we expect the elements of $can(\mathbf{α})$ to behave as independent Gaussian random variables. Again, we have the classical estimate

$$\Pr\Big[|can(\mathbf{α})_i| > c \cdot \sigma^{can}\Big] \le erfc(c/\sqrt{2}),$$

where σ^{can} is the standard deviation of the elements of $can(\mathbf{α})$. Applying the Union Bound to obtain an upper bound on $\|\mathbf{α}\|^{can} = \max_i |can(\mathbf{α})_i|$ we find the estimate

$$\Pr\Big[\|\mathbf{α}\|^{can} > c \cdot \sigma^{can}\Big] \le erfc(c/\sqrt{2}) \cdot N/2,$$

The division by two here is because the M–th roots of unity come in complex conjugate pairs. Translating back to the standard deviation on the polynomial embedding this becomes

$$\Pr\Big[\|\mathbf{α}\|^{can} > c \cdot \sigma \cdot \sqrt{N}\Big] \le erfc(c/\sqrt{2}) \cdot N/2,$$

In any application we will have the bound $N \le 2^{20}$, thus to obtain a right hand side of the above equation to be less than $err = 2^{-128}$ we can set,

$$c_{err,N} = 14.10$$

as a fixed constant for c for a ring of dimension N. This simplifies the analysis somewhat, with smaller values of N one can obviously select smaller values of $c_{err,N}$ if one wanted to.

Suppose we have a term of the form $\mathbf{α} \odot \mathbf{b}$ for $\mathbf{α}, \mathbf{b} \in \mathbf{R}_Q$, then we would also like to probabilistically bound $\|\mathbf{α} \odot \mathbf{b}\|^{can}$ in terms of the standard deviations σ_a and σ_b of the coefficients in the polynomial embedding of **α** and **b**. The probability density function of the product of two Gaussian variables is given by

$$p(x) = \frac{K_0(\frac{|x|}{\sigma_a \cdot \sigma_b})}{\pi \cdot \sigma_a \cdot \sigma_b}$$

where $K_n(z)$ is the modified Bessel function of the second kind. Using a known lower bound (see for

example [YC17]) $K_0(z) > \frac{\sqrt{\pi}e^{-z}}{\sqrt{2 \cdot (z + \frac{1}{4})}}$ we obtain

$$p(x) > \frac{e^{-\frac{|x|}{\sigma_a \cdot \sigma_b}}}{\sqrt{2\pi(|x| + \sigma_a\sigma_b/4)\sigma_a\sigma_b}}. \tag{2}$$

Due to equation (2) the following Lemma is seen to provide a quick, yet relatively tight, rule of thumb for product distributions.

Lemma 2. *Let $\mathbf{a}, \mathbf{b} \in \mathbf{R}_Q$ have standard deviations of the coefficients in the polynomial embedding given by σ_a and σ_b then, assuming $N \leq 2^{20}$,*

$$\Pr\left[\|\mathbf{a} \odot \mathbf{b}\|^{can} < c_{err,N}^2 \cdot \sigma_a \cdot \sigma_b \cdot N\right] \leq \approx err = 2^{-128}.$$

Proof. Now we know $\|\mathbf{a} \odot \mathbf{b}\|^{can} \leq \|\mathbf{a}\|^{can} \cdot \|\mathbf{b}\|^{can}$, thus we have

$$\Pr\left[\|\mathbf{a} \odot \mathbf{b}\|^{can} \leq c_{err,N}^2 \cdot \sigma_a \cdot \sigma_b \cdot N\right] \geq \Pr\left[\|\mathbf{a}\|^{can} \leq c_{err,N} \cdot \sigma_a \cdot \sqrt{N}\right] \cdot \Pr\left[\|\mathbf{b}\|^{can} \leq c_{err,N} \cdot \sigma_b \cdot \sqrt{N}\right]$$

$$\geq \left(1 - erfc(c_{err,N}/\sqrt{2}) \cdot N/2\right)^2$$

$$\geq \approx 1 - 2^{-128} \quad \text{if } N \leq 2^{20}.$$

$\square$

Note, the above bound in Lemma 2 on a product is not the same as that used in [GHS12]; this latter reference has errors in it.

5.4.3 Ciphertext "Noise"

A Ring–LWE ciphertext will come in one of two forms, depending on whether we place the message $\mathbf{m} \in \mathbf{R}_P$ in the "top" or the "bottom" of the ciphertext range, i.e. we either have a ciphertext of the form

$$(\mathbf{a}, \mathbf{b} = \mathbf{a} \odot \mathbf{s} + P \cdot \mathbf{e} + \mathbf{m}) \in \mathbf{R}_Q^2$$

or of the form

$$(\mathbf{a}, \mathbf{b} = \mathbf{a} \odot \mathbf{s} + \mathbf{e} + \Delta \cdot \mathbf{m}) \in \mathbf{R}_Q^2,$$

where recall in the last equation we have $\Delta = \lfloor Q/P \rfloor$. We call these Type–I and Type–II.

A standard LWE ciphertext (in this document) will always be of Type–II, i.e. we have

$$(\mathbf{a}, b = \mathbf{a} \cdot \mathbf{s} + e + \Delta \cdot m) \in (\mathbb{Z}/(Q))^{N+1}.$$

The decryption of a ciphertext consists of first performing the operation

$$\mathbf{p} = \mathbf{b} - \mathbf{a} \odot \mathbf{s} \pmod{Q} \quad (\text{resp. } p = b - \mathbf{a} \cdot \mathbf{s} \pmod{Q}),$$

to produce what is called the **pre-decryption $\mathbf{p}$** (resp. p). Then the message is obtained by decoding the value $\mathbf{p}$ (resp. p); either by reduction modulo P in case of Type–I ciphertexts, or by rounding on division by Δ in case of Type–II ciphertexts.

Correct decryption is guaranteed as long as the pre-decryption $\mathbf{p}$ does not wrap around. In particular we require, for correct decryption, that

$$\|\mathbf{p}\|_\infty \leq Q/2$$

in the case of Type–I ciphertexts, or

$$\|\mathbf{p} - \Delta \cdot \mathbf{m}\|_\infty \leq Q/(2 \cdot P) \quad (\text{resp. } \|p - \Delta \cdot m\| \leq Q/(2 \cdot P))$$

in the case of Type–II ciphertexts. Note, in the case where P divides Q (which happens in TFHE) we have that $Q/(2 \cdot P) = \Delta/2$.

A standard (but crude) noise analysis technique for Ring–LWE based systems is performed as follows (and one which will be used for BGV and BFV in our document, for TFHE we perform a slightly different analysis):

1. One maintains with each ciphertext a bound B on the canonical norm of the error term associated to each ciphertext. In the context of Type–I this is the pre–decryption value, and in the context of Type–II ciphertexts this is a bound on the canonical norm of $\mathbf{p} - \Delta \cdot \mathbf{m}$ (resp. $p - \Delta \cdot m$).
2. To ensure decryption is valid we need to ensure that

$$E_M \cdot B \le Q/2,$$

for Type–I ciphertexts, or

$$E_M \cdot B \le Q/(2 \cdot P),$$

for Type–II ciphertexts; where E_M is the ring constant from earlier.
3. When performing an FHE operation we update the value B using the estimates from above in relation to the $erfc$ function, for the associated random variables which are added onto B, plus any scaling of B by constants.

5.5 Modulus Switching of Ring–LWE Ciphertexts

An important operation in our methodology for threshold decryption and key generation for schemes such as BGV and BFV is that of modulus switching. We will only utilize modulus switching in our applications later for Type–I ciphertexts; so we only explain the methodology in this situation.

Suppose we have a ciphertext $(\mathbf{a}, \mathbf{b})$ with plaintext/ciphertext moduli P/Q. We wish to switch to a new modulus q, a process which we will denote by

$$ModSwitch^{Q \to q}(\mathbf{a}, \mathbf{b}).$$

We assume that the secret key $\mathbf{s} \in \mathbf{R}_Q$ is chosen so each component of $can(\mathbf{s})$ has zero expectation, and whose coefficients in the polynomial embedding are bounded by $\min(q, Q)/2$ (this is to ensure that the secret key is defined modulo q and Q. In practice this is a non–assumption as the secret key will be chosen to be a polynomial whose coefficients are bounded in absolute value by one).

To ease exposition we assume that $q \equiv Q \pmod{P}$; this latter restriction is not going to pose any problems in practice. This restriction can be removed by using so–called "correction factors" which are held with each ciphertext value; see [HS20] for details. Without the use of such correction factors we make the following design decision which is enough for our purposes

> **Design Decision 26: Restrictions on Q for BGV Modulus Switching**
>
> We assume, for BGV and BFV, that the ciphertext modulus Q is a product of primes $Q = Q_1 \cdots Q_L$, where the primes Q_i and the extra large prime R for key switching (see later) are chosen to satisfy $Q_i \equiv R \equiv 1 \pmod{P}$ for all i.

We start with a ciphertext $(\mathbf{a}, \mathbf{b}) \in \mathbf{R}_Q^2$ which satisfies

$$\mathbf{b} = \mathbf{a} \odot \mathbf{s} + P \cdot \mathbf{e} + \mathbf{m} \pmod{Q},$$

and we aim to output a ciphertext $(\mathbf{a}', \mathbf{b}') \in \mathbf{R}_q^2$ which satisfies

$$\mathbf{b}' = \mathbf{a}' \odot \mathbf{s} + P \cdot \mathbf{e}' + \mathbf{m} \pmod{q},$$

We follow the method of Section 4.1 of [HS20], which we overview in Figure 7.

ModSwitch

$ModSwitch^{Q\rightarrow q}(\mathbf{a}, \mathbf{b})$:
1. Think of $\mathbf{a}, \mathbf{b} \in \mathbf{R}$ and not in $\mathbf{R}_Q$.
2. $\overline{\mathbf{a}} \leftarrow \lceil \mathbf{a} \cdot q/Q \rfloor$, $\overline{\mathbf{b}} \leftarrow \lceil \mathbf{b} \cdot q/Q \rfloor$.
3. $\mathbf{d}_a \leftarrow q \cdot \mathbf{a} - Q \cdot \overline{\mathbf{a}}$, $\mathbf{d}_b \leftarrow q \cdot \mathbf{b} - Q \cdot \overline{\mathbf{b}}$.
4. $\mathbf{e}_a \leftarrow \mathbf{d}_a/Q \pmod{P}$, $\mathbf{e}_b \leftarrow \mathbf{d}_b/Q \pmod{P}$.
   As $Q \equiv 1 \pmod{P}$ *due to Design Decision 26 this is simply a reduction modulo P.*
5. $\mathbf{f}_a \leftarrow \overline{\mathbf{a}} + \mathbf{e}_a$, $\mathbf{f}_b \leftarrow \overline{\mathbf{b}} + \mathbf{e}_b$.
6. $\mathbf{a}' \leftarrow \mathbf{f}_a \pmod{q}$, $\mathbf{b}' \leftarrow \mathbf{f}_b \pmod{q}$.
7. Return $(\mathbf{a}', \mathbf{b}')$.

Figure 7: $ModSwitch^{Q\rightarrow q}(\mathbf{a}, \mathbf{b})$ for Type–I Ring–LWE Ciphertexts.

Lemma 3. *If B_{input} is a bound which holds with probability $1 - err$ on the canonical norm of the pre–decryption of the input ciphertext $(\mathbf{a}, \mathbf{b})$ to $ModSwitch^{Q\rightarrow q}(\mathbf{a}, \mathbf{b})$ in Figure 7, and $\sigma_{\mathfrak{sf}}$ is the standard deviation of the coefficients of the polynomial embedding of the secret key, we can then bound the canonical norm of the pre–decryption of the output ciphertext $(\mathbf{a}', \mathbf{b}')$ by*

$$B_{output} = \frac{q \cdot B_{input}}{Q} + c_{err,N} \cdot (P+1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sigma_{\mathfrak{sf}} \cdot \sqrt{N}) \tag{3}$$

with probability at least $1 - err$. Thus the algorithm is correct, with probability $1 - err$, assuming $E_M \cdot B_{output} < Q/2$.

Proof. We first show (potential) correctness, and then we explain how this modulus switching operation affects the noise (and the bound on B_{output} needed to guarantee correctness).
Correctness Denote the pre–decryption of $(\mathbf{a}, \mathbf{b})$ over $\mathbf{R}$ (i.e. before reduction modulo Q) by

$$\mathbf{p} = \mathbf{b} - \mathbf{a} \odot \mathbf{s} - Q \cdot \mathbf{g}.$$

Now set

$$\mathbf{p}' = \mathbf{f}_b - \mathbf{f}_a \odot \mathbf{s} - q \cdot \mathbf{g}$$

for the same value $\mathbf{g}$. We see that, for some $\mathbf{d}_a, \mathbf{e}_a, \mathbf{f}_a$ etc,

$$q \cdot \mathbf{a} = Q \cdot \overline{\mathbf{a}} + \mathbf{d}_a \equiv Q \cdot (\overline{\mathbf{a}} + \mathbf{e}_a) = Q \cdot \mathbf{f}_a \pmod{P},$$
$$q \cdot \mathbf{b} = Q \cdot \overline{\mathbf{b}} + \mathbf{d}_b \equiv Q \cdot (\overline{\mathbf{b}} + \mathbf{e}_b) = Q \cdot \mathbf{f}_b \pmod{P}.$$

Hence,

$$q \cdot \mathbf{p} \equiv Q \cdot \mathbf{f}_b - Q \cdot \mathbf{f}_a \odot \mathbf{s} - q \cdot Q \cdot \mathbf{g} \pmod{P},$$
$$\equiv Q \cdot (\mathbf{f}_b - \mathbf{f}_a \odot \mathbf{s} - q \cdot \mathbf{g}) \pmod{P}$$
$$\equiv Q \cdot \mathbf{p}' \pmod{P}.$$

Thus the ciphertext $(\mathbf{a}', \mathbf{b}')$ will decrypt to the correct value as long as the "noise" value $\mathbf{p}'$ is sufficiently small, since

$$\mathbf{p}' \equiv \frac{q \cdot \mathbf{p}}{Q} \equiv \mathbf{p} \pmod{P}.$$

Noise Analysis We need to show there is no wrap around in computing $\mathbf{p}'$, and hence $\mathbf{p}'$ **is** the pre–

decryption of $(\boldsymbol{\alpha}', \mathbf{b}')$ over $\mathbf{R}$. We have over $\mathbf{R}$

$$
\begin{aligned}
\mathbf{p}' &= \mathbf{f}_b - \mathbf{f}_a \odot \mathbf{s} - q \cdot \mathbf{g} \\
&= (\overline{\mathbf{b}} + \mathbf{e}_b) - (\overline{\boldsymbol{\alpha}} + \mathbf{e}_a) \odot \mathbf{s} - q \cdot \mathbf{g} \\
&= \left(\frac{q}{Q} \cdot \mathbf{b} - \frac{\mathbf{d}_b}{Q} + \mathbf{e}_b \right) - \left(\frac{q}{Q} \cdot \boldsymbol{\alpha} - \frac{\mathbf{d}_a}{Q} + \mathbf{e}_a \right) \odot \mathbf{s} - q \cdot \mathbf{g} \\
&= \frac{q}{Q} \cdot \mathbf{p} + Q \cdot \frac{q}{Q} \cdot \mathbf{g} - \left(\frac{\mathbf{d}_b}{Q} - \mathbf{e}_b \right) + \left(\frac{\mathbf{d}_a}{Q} - \mathbf{e}_a \right) \odot \mathbf{s} - q \cdot \mathbf{g} \\
&= \frac{q}{Q} \cdot \mathbf{p} + \mathbf{h}_b - \mathbf{h}_a \odot \mathbf{s}.
\end{aligned}
$$

where we write

$$
\mathbf{h}_a = \mathbf{e}_a - \frac{\mathbf{d}_a}{Q} \text{ and } \mathbf{h}_b = \mathbf{e}_b - \frac{\mathbf{d}_b}{Q}.
$$

Note the coefficients of $\mathbf{d}_a$, $\mathbf{d}_b$ lie in the interval $[-Q/2, \ldots, Q/2]$, and the coefficients of $\mathbf{e}_a$, $\mathbf{e}_b$ lie in the interval $[-P/2, \ldots, P/2]$. We make the assumption that these coefficients are distributed as uniformly random variables on these intervals[10].

If B_{input} is a bound which holds with probability $1 - err$ on the canonical norm of the pre-decryption of the input ciphertext $(\boldsymbol{\alpha}, \mathbf{b})$, and $\sigma_{\mathfrak{sk}}$ is the standard deviation of the coefficients of the polynomial embedding of the secret key, we can then bound the canonical norm of the pre-decryption of the output ciphertext by

$$
\begin{aligned}
B_{output} &= \| \frac{q}{Q} \cdot \mathbf{p} + \mathbf{h}_b - \mathbf{h}_a \odot \mathbf{s} \|^{can} \\
&\leq \| \frac{q}{Q} \cdot \mathbf{p} \|^{can} + \| \mathbf{h}_b \|^{can} + \| \mathbf{h}_a \odot \mathbf{s} \|^{can} \\
&\leq \frac{q \cdot B_{input}}{Q} + \| \mathbf{e}_b - \frac{\mathbf{d}_b}{Q} \|^{can} + \| (\mathbf{e}_a - \frac{\mathbf{d}_a}{Q}) \odot \mathbf{s} \|^{can} \\
&\leq \frac{q \cdot B_{input}}{Q} + c_{err,N} \cdot \left(P \cdot \sqrt{N/12} + \frac{Q \cdot \sqrt{N/12}}{Q} \right) \cdot (1 + c_{err,N} \cdot \sigma_{\mathfrak{sk}} \cdot \sqrt{N}) \\
&= \frac{q \cdot B_{input}}{Q} + c_{err,N} \cdot (P + 1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sigma_{\mathfrak{sk}} \cdot \sqrt{N})
\end{aligned}
$$

with probability at least $1 - err$. $\qquad\square$

Thus, apart from a "small" additive term being added into the noise, the noise has been multiplied by a factor of q/Q; when $Q \gg q$ this means the noise has been squashed somewhat. We write, for usage later,

$$
B_{Scale} = c_{err,N} \cdot (P + 1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sigma_{\mathfrak{sk}} \cdot \sqrt{N}).
$$

So our about formulae becomes

$$
B_{output} = \frac{q \cdot B_{input}}{Q} + B_{Scale}.
$$

5.6 Fully Homomorphic Encryption

5.6.1 BGV

In this section we give a brief overview of the BGV [BGV12] scheme. We do not provide a full optimized specification, as that is beyond the scope of this document. We focus instead on what is needed to understand how our threshold schemes work in the context of BGV. Note, due to the fact that our secret

---

[10] See [HS2O] for a discussion on when and why these assumptions are valid.

keys are generated inside a threshold protocol some of the optimizations presented in say [HS20] are not applicable.

A ciphertext modulus in BGV is a product of moduli $Q = Q_1 \cdots Q_L$, each one corresponding (roughly) to a level of multiplications. As we progress through a computation the levels are peeled off, and our ciphertext modulus decreases. We will write $Q = Q^{(L)}$ and $Q^{(\ell)} = Q_1 \cdots Q_\ell$ to be the ciphertext modulus at level ℓ. We start at level L and then decrease levels until we obtain level one.

Following Design Decision 16, Design Decision 24 and Design Decision 26, to simplify analysis we do not consider the following aspects of BGV.

1. We present BGV as a leveled scheme, and do not describe any form of bootstrapping. Bootstrapping may be added to the scheme (if the number of levels allows this), but it is not important for our threshold protocols in the case of BGV[11]. Thus our presentation of BGV is actually as a *Somewhat* Homomorphic Encryption scheme only. Bootstrapping in BGV is considered costly without hardware acceleration, and so many applications avoid it.

2. We do not consider plaintext correction factors. So in particular we require $Q_i \equiv 1 \pmod{P}$ for all i, where P is the plaintext modulus. This implies $Q^{(i)} \equiv Q^{(j)} \equiv 1 \pmod{P}$ for all i, j. Relaxing the requirement on the Q_i can be done by introducing correction factors as explained in [HS20].

3. We do not consider slot operations, i.e. permutations between slots. This is easily added to the description if needed, see [HS20] for details.

The above optimizations can be added as required, with little (if any) changes to our threshold protocols.

5.6.1.1 The BGV Scheme Description: The parameters associated with our BGV scheme are as follows (P, Q, N, L, R, B), where:

- The plaintext modulus P.
- The top ciphertext modulus $Q = Q_1 \cdots Q_L$.
- The values Q_i are prime with $Q_i \equiv 1 \pmod{2 \cdot N}$.
- The cyclotomic ring $\mathbf{R}^{(M)}$, and its degree $N = \phi(M)$.
- The supported number of levels L.
- A large prime R used for key–switching. We also choose $R \equiv Q_i \equiv 1 \pmod{P}$ to avoid some complications in the methods below, as well as $R \equiv 1 \pmod{2 \cdot N}$.
- A distribution $NewHope(N, B)$ to generate noise values, for a very small value of B (typically $B = 1$). Recall, this is chosen to enable efficient key generation within an MPC engine[12].

A BGV ciphertext, at level ℓ, will consist of a tuple

$$\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{\mathfrak{ct}})$$

where $\mathbf{c}_0, \mathbf{c}_1 \in \mathbf{R}_{Q^{(\ell)}}$ and $B_{\mathfrak{ct}}$ is a bound on the canonical norm of the pre–decryption value, i.e. $\mathbf{c}_0 - \mathfrak{sk} \odot \mathbf{c}_1$, which holds with probability at least $1 - err$. The ciphertext will encrypt a message $\mathbf{m} \in \mathbf{R}_P$.

A BGV secret key will consist of a vector $\mathfrak{sk} \in \mathbf{R}$ drawn from the distribution $NewHope(N, 1)$. The standard deviation of the coefficients of such a distribution is $\sigma_{\mathfrak{sk}} = 1/\sqrt{2}$. This distribution is selected for our purposes as it is easy to generate within an MPC–engine (see later), i.e. for much the same reason as we selected $NewHope(N, B)$ as the noise distribution.

The main algorithms underlying BGV can now be defined as in Figure 8, Figure 9 and Figure 10. The key generation produces two Ring-LWE samples. One $(\mathfrak{pk}_a, \mathfrak{pk}_b)$ in the ring $\mathbf{R}_Q$ which is used for encryption, and one $(\mathfrak{pk}'_a, \mathfrak{pk}'_b)$ in a larger ring $\mathbf{R}_{R \cdot Q}$ which is used for key–switching in the multiplication routines. For ease of notation, when explaining our ZKPoKs later, we will let $A_{\mathfrak{pk}}$ (resp. $B_{\mathfrak{pk}}$) denote the matrix

---

[11]Bootstrapping is more fundamental to TFHE, so when we get to TFHE we will specify how to bootstrap

[12]For public key encryption it is also useful as it enables efficient zero-knowledge proof's of correctness of the underlying ciphertext; see later.

representation of the ring element $\mathfrak{pk}_a$ (resp. $\mathfrak{pk}_b$). With this notation we have that

$$\mathfrak{pk}_a \odot \mathbf{v} = A_{\mathfrak{pk}} \cdot \mathbf{v}.$$

BGV – Part I

BGV.KeyGen(N, Q, P, B, R):
1. $\mathfrak{sk} \leftarrow NewHope(N, 1)$, i.e. each element of the N coefficients are selected from $\{-1, 0, 1\}$ with probabilities $\Pr[0] = 1/2$ and $\Pr[1] = \Pr[-1] = 1/4$.
2. $\mathfrak{pk}_a \leftarrow \mathbf{R}_Q$.
3. $\mathbf{e}_{\mathfrak{pk}} \leftarrow NewHope(N, B)$.
4. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_a \odot \mathfrak{sk} + P \cdot \mathbf{e}_{\mathfrak{pk}} \in \mathbf{R}_Q$.
5. Set $T \leftarrow R \cdot Q$.
6. $\mathfrak{pk}_a' \leftarrow \mathbf{R}_T$.
7. $\mathbf{e}_{\mathfrak{pk}}' \leftarrow NewHope(N, B)$.
8. $\mathfrak{pk}_b' \leftarrow \mathfrak{pk}_a' \odot \mathfrak{sk} + P \cdot \mathbf{e}_{\mathfrak{pk}}' - R \cdot \mathfrak{sk} \odot \mathfrak{sk} \in \mathbf{R}_T$.
9. $\mathfrak{pk} \leftarrow \{(\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}_a', \mathfrak{pk}_b')\}$.
10. Return $(\mathfrak{pk}, \mathfrak{sk})$.

BGV.Enc$(\mathbf{m}, \mathfrak{pk})$:
1. $\mathbf{v} \leftarrow NewHope(N, 1)$.
2. $\mathbf{e}_0, \mathbf{e}_1 \leftarrow NewHope(N, B)$.
3. $\mathbf{c}_0 \leftarrow \mathfrak{pk}_b \odot \mathbf{v} + P \cdot \mathbf{e}_0 + \mathbf{m} \in \mathbf{R}_Q$.
4. $\mathbf{c}_1 \leftarrow \mathfrak{pk}_a \odot \mathbf{v} + P \cdot \mathbf{e}_1 \in \mathbf{R}_Q$.
5. $B_{\mathfrak{ct}} \leftarrow c_{err,N} \cdot P \cdot \left(\sqrt{N/12} + \sqrt{B} \cdot (c_{err,N} \cdot N + \sqrt{N/2}) \right)$.
6. Return $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, L, B_{\mathfrak{ct}})$, In some applications we will require the return of $((\mathbf{c}_0, \mathbf{c}_1, L, B_{\mathfrak{ct}}), (\mathbf{v}, \mathbf{e}_0, \mathbf{e}_1))$ in order to prove knowledge and correctness of the en-cryption via the ZKPoKs in Section 7.6.

BGV.Dec$(\mathfrak{ct}, \mathfrak{sk})$:
1. Write $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{\mathfrak{ct}})$.
2. If $E_M \cdot B_{\mathfrak{ct}} > Q^{(\ell)}/2$ then *abort* (this ciphertext has too much noise).
3. $\mathbf{p} \leftarrow \mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk} \pmod{Q^{(\ell)}}$.
4. $\mathbf{m} \leftarrow \mathbf{p} \pmod{P}$.
5. Return $\mathbf{m}$.

Figure 8: The BGV Leveled Homomorphic Encryption Scheme – Part I.

Lemma 4. *In the algorithm BGV.Enc$(\mathbf{m}, \mathfrak{pk})$ in Figure 8, the value of $B_{\mathfrak{ct}}$ is a bound on the canonical norm of the pre-decryption value of the output (honestly generated) ciphertext which holds with probability at least $1 - err$.*

Proof. This follows from the inequalities

$$\|\mathbf{c}_0 - \mathfrak{sk} \odot \mathbf{c}_1\|^{can} = \|((\mathfrak{pk}_a \odot \mathfrak{sk} + P \cdot \mathbf{e}_{\mathfrak{pk}}) \odot \mathbf{v} + P \cdot \mathbf{e}_0 + \mathbf{m}) - (\mathfrak{pk}_a \odot \mathbf{v} + P \cdot \mathbf{e}_1) \odot \mathfrak{sk}\|^{can}$$

$$= \|\mathbf{m} + P \cdot (\mathbf{e}_{\mathfrak{pk}} \odot \mathbf{v} + \mathbf{e}_0 - \mathbf{e}_1 \odot \mathfrak{sk})\|^{can}$$

$$\leq \|\mathbf{m}\|^{can} + P \cdot \left(\|\mathbf{e}_{\mathfrak{pk}} \odot \mathbf{v}\|^{can} + \|\mathbf{e}_0\|^{can} + \|\mathbf{e}_1 \odot \mathfrak{sk}\|^{can} \right)$$

$$\leq c_{err,N} \cdot \left(P \cdot \sqrt{N/12} \right.$$
$$\left. + P \cdot \left(c_{err,N} \cdot \sqrt{B \cdot N/2} \cdot \sqrt{N/2} + \sqrt{B \cdot N/2} + c_{err,N} \cdot \sqrt{B \cdot N/2} \cdot \sqrt{N/2} \right) \right)$$

$$= c_{err,N} \cdot P \cdot \left(\sqrt{N/12} + \sqrt{B} \cdot (c_{err,N} \cdot N + \sqrt{N/2}) \right)$$

Note, this assumes that the encryptor is honest, i.e. they are encrypting by generating the terms $\mathbf{v}$, $\mathbf{e}_0$ and $\mathbf{e}_1$ according to the correct distributions. A malicious encryptor may encrypt using the extremes of the distributions. We can always force the encryptor to encrypt with at worst these extremes, by requiring each encryption to come with a zero–knowledge proof of correct encryption (as in Section 7.6).

If we assume a worst case bound on the encryption noise, but assuming an average case analysis of the noise terms in the key generation algorithm (i.e. we assume the key generation is performed honestly, or equivalently via our threshold protocols), then we obtain a ciphertext noise bound of roughly the same size (effectively it will increase by a factor of $\sqrt{12}$) as the same law–of–large numbers and union–bound will ensure that the maximum of a sum of N complex numbers (needed to compute the canonical norm) will result in roughly the same statistical analysis.

BGV – Part II

$BGV.Scale(\mathfrak{ct}, \ell')$:
    This transforms a ciphertext to a new lower level ℓ'.
      1. Write $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{input})$.
      2. If $\ell = \ell'$ then return $\mathfrak{ct}$.
      3. If $\ell < \ell'$ or $\ell < 1$ then *abort*.
      4. $(\mathbf{c}_1', \mathbf{c}_0') \leftarrow ModSwitch^{Q^{(\ell)} \rightarrow Q^{(\ell')}}(\mathbf{c}_1, \mathbf{c}_0)$.
      5. Set B_{output} as per equation (3), with $\sigma_{\mathfrak{st}} = 1/\sqrt{2}$, $q = Q^{(\ell')}$ and $Q = Q^{(\ell)}$, i.e.

$$B_{output} \leftarrow \frac{Q^{(\ell')} \cdot B_{input}}{Q^{(\ell)}} + B_{Scale}.$$

      6. Return $(\mathbf{c}_0', \mathbf{c}_1', \ell', B_{output})$.

$BGV.KeySwitch((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_{input})$:
      1. $T^{(\ell)} \leftarrow R \cdot Q^{(\ell)}$.
      2. $\mathbf{c}_0' \leftarrow R \cdot \mathbf{d}_0 + \mathfrak{pk}_b' \odot \mathbf{d}_2 \pmod{T^{(\ell)}}$.
      3. $\mathbf{c}_1' \leftarrow R \cdot \mathbf{d}_1 + \mathfrak{pk}_a' \odot \mathbf{d}_2 \pmod{T^{(\ell)}}$.
      4. $(\mathbf{c}_1, \mathbf{c}_0) \leftarrow ModSwitch^{T^{(\ell)} \rightarrow Q^{(\ell)}}(\mathbf{c}_1', \mathbf{c}_0')$.
      5. Set

$$B_{output} \leftarrow B_{input} + \frac{B_{KeySwitch} \cdot Q^{(\ell)}}{R} + B_{Scale}.$$

      6. Return $(\mathbf{c}_0, \mathbf{c}_1, \ell, B_{output})$.

Figure 9: The BGV Leveled Homomorphic Encryption Scheme – Part II.

The key switching procedure $BGV.KeySwitch((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_{input}$ in Figure 9 maps a ciphertext which decrypts via

$$\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{st} - \mathbf{d}_2 \odot \mathfrak{st} \odot \mathfrak{st} \equiv \mathbf{m} + P \cdot \mathbf{e} \pmod{Q^{(\ell)}}$$

into one which decrypts via the usual equation, where on input we are guaranteed that

$$\|\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{st} - \mathbf{d}_2 \odot \mathfrak{st} \odot \mathfrak{st}\| \leq B_{input}.$$

The methodology used is the one presented in [GHS12]. That the methodology is correct follows from the following Lemma:

Lemma 5. *The algorithm $BGV.KeySwitch((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_{input})$ in Figure 9 is correct with probability $1 - err$.*

Proof. To see the correctness of this procedure first notice that

$$
\begin{aligned}
\mathbf{c}'_0 - \mathfrak{sk} \odot \mathbf{c}'_1 &= R \cdot \mathbf{d}_0 + \mathfrak{pk}'_b \odot \mathbf{d}_2 - \mathfrak{sk} \odot (R \cdot \mathbf{d}_1 + \mathfrak{pk}'_a \odot \mathbf{d}_2) \\
&= R \cdot \mathbf{d}_0 + (\mathfrak{pk}'_a \odot \mathfrak{sk} + P \cdot \mathbf{e}'_{\mathfrak{pk}} - R \cdot \mathfrak{sk} \odot \mathfrak{sk}) \odot \mathbf{d}_2 - \mathfrak{sk} \odot (R \cdot \mathbf{d}_1 + \mathfrak{pk}'_a \odot \mathbf{d}_2) \\
&= R \cdot (\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{sk} - \mathbf{d}_2 \odot \mathfrak{sk} \odot \mathfrak{sk}) + \mathfrak{pk}'_a \odot \mathfrak{sk} \odot \mathbf{d}_2 + P \cdot \mathbf{e}'_{\mathfrak{pk}} \odot \mathbf{d}_2 - \mathfrak{sk} \odot \mathfrak{pk}'_a \odot \mathbf{d}_2 \\
&= R \cdot (\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{sk} - \mathbf{d}_2 \odot \mathfrak{sk} \odot \mathfrak{sk}) + P \cdot \mathbf{e}'_{\mathfrak{pk}} \odot \mathbf{d}_2 \quad (\text{mod } T^{(\ell)}) \\
&= R \cdot (\mathbf{m} + P \cdot \mathbf{e}) + P \cdot \mathbf{e}'_{\mathfrak{pk}} \odot \mathbf{d}_2 \quad (\text{mod } T^{(\ell)})
\end{aligned}
$$

Assuming both LHS and RHS are less than $T^{(\ell)}$ we get an equality over $\mathbf{R}$ (and not just $\mathbf{R}_{T^{(\ell)}}$). And so modulo P we will obtain the correct plaintext as $R \equiv Q^{(\ell)} \equiv 1 \pmod P$, after we apply the scaling operation.

It remains to analyze the added noise, and so derive a formulae for $B_{KeySwitch}$. We assume that $\mathbf{d}_2$ behaves like a random polynomial with coefficients drawn uniformly from $(-Q^{(\ell)}/2, \ldots, Q^{(\ell)}/2)$. Thus $can(\mathbf{d}_2)$ has entries which are close to a Gaussian with standard deviation $Q^{(\ell)} \cdot \sqrt{N/12}$. We can assume that $can(\mathbf{e}'_{\mathfrak{pk}})$ has Gaussian like entries with standard deviation $\sqrt{B \cdot N/2}$. Thus we have with probability at least $1 - err$

$$
\|P \cdot \mathbf{e}'_{\mathfrak{pk}} \odot \mathbf{d}_2\|^{can} \le c^2_{err,N} \cdot P \cdot \sqrt{B/24} \cdot Q^{(\ell)} \cdot N = B_{KeySwitch} \cdot Q^{(\ell)},
$$

where

$$
B_{KeySwitch} = c^2_{err,N} \cdot P \cdot \sqrt{B/24} \cdot N.
$$

From which we derive the bound used in Figure 9

$$
\begin{aligned}
\|\mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk}\|^{can} &\le \frac{1}{R} \cdot \|\mathbf{c}'_0 - \mathfrak{sk} \odot \mathbf{c}'_1\|^{can} + B_{Scale} \\
&\le \frac{1}{R} \cdot \left(\|R \cdot (\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{sk} - \mathbf{d}_2 \odot \mathfrak{sk} \odot \mathfrak{sk})\|^{can} + \|P \cdot \mathbf{e}'_{\mathfrak{pk}} \odot \mathbf{d}_2\|^{can} \right) + B_{Scale} \\
&\le B_{input} + \frac{B_{KeySwitch} \cdot Q^{(\ell)}}{R} + B_{Scale}.
\end{aligned}
$$

$\square$

Note, in the *Add* routine of Figure 10 one can avoid the *Scale* operations and just perform modular reduction if the noise magnitude is not too large, see the Appendix B.5 of the full version of [GHS12] for details. In the scalar and general *Mult* routines we apply modulus reduction to ensure the noise on entering the multiplication is bounded by $\lambda \cdot B_{Mult}$ for two constants λ and B_{Mult} to be determined later.

5.6.1.2 The BGV Scheme Correctness: The constants λ and B_{Mult}, used in the routine *Mult*, are the key to the eventual correctness of the BGV scheme, and the derivation of valid parameters. Assume that we execute circuits which consists of layers of multiplication gates, followed by layers of addition gates, where we allow fan in two for the multiplication gates and fan in λ for the addition gates. Also assume that the final layer is a layer of multiplication gates.

We wish to maintain the invariant that the output of the *Mult* operation is a ciphertext which has noise bound less than B_{Mult}. This leads to two observations:

1. A fresh ciphertext can have noise B_{ct} much larger than B_{Mult}, but it can be brought down to below $\lambda \cdot B_{Mult}$ by applying a *Scale* operation, which will consume a level, at the start of the first multiplication operation.

2. For such circuits as described above, the first two lines of the algorithm *Mult* are no–op's if the output of *Scale* if our invariant is satisfied and the input ciphertexts are the output of an addition layer with fan in λ.

BGV – Part III

$BGV.Add(\mathfrak{ct}_a, \mathfrak{ct}_b)$:
1. $\ell \leftarrow \min(\ell_a, \ell_b)$, where ℓ_a and ℓ_b are the levels of the ciphertexts $\mathfrak{ct}_a$ and $\mathfrak{ct}_b$.
2. $\mathfrak{ct}_a \leftarrow BGV.Scale(\mathfrak{ct}_a, \ell)$.
3. $\mathfrak{ct}_b \leftarrow BGV.Scale(\mathfrak{ct}_b, \ell)$.
4. Write $\mathfrak{ct}_a = (\mathbf{c}_0^a, \mathbf{c}_1^a, \ell_a, B_a)$.
5. Write $\mathfrak{ct}_b = (\mathbf{c}_0^b, \mathbf{c}_1^b, \ell_b, B_b)$.
6. $\mathbf{c}_0 \leftarrow \mathbf{c}_0^a + \mathbf{c}_0^b \pmod{Q^{(\ell)}}$.
7. $\mathbf{c}_1 \leftarrow \mathbf{c}_1^a + \mathbf{c}_1^b \pmod{Q^{(\ell)}}$.
8. $B \leftarrow B_a + B_b$.
9. $\mathfrak{ct} \leftarrow (\mathbf{c}_0, \mathbf{c}_1, B)$.
10. Return $\mathfrak{ct}$.

$BGV.Mult(\alpha, \mathfrak{ct})$:
This is the scalar multiplication routine, for $\alpha \in \mathbf{R}_P$.
1. Write $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, B)$.
2. While $B > \lambda \cdot B_{Mult}$ do $\mathfrak{ct} \leftarrow BGV.Scale(\mathfrak{ct}, \ell_a - 1)$, i.e. keep decreasing the level until the noise bound in $\mathfrak{ct}$ is less than $\lambda \cdot B_{Mult}$. Denote the output also by $(\mathbf{c}_0, \mathbf{c}_1, B)$.
3. $\mathbf{c}_0' \leftarrow \alpha \cdot \mathbf{c}_0$.
4. $\mathbf{c}_1' \leftarrow \alpha \cdot \mathbf{c}_1$.
5. $B' \leftarrow \|\alpha\|^{can} \cdot B$.
6. $\mathfrak{ct}' \leftarrow (\mathbf{c}_0', \mathbf{c}_1', B')$.
7. Return $\mathfrak{ct}$.

$BGV.Mult(\mathfrak{ct}_a, \mathfrak{ct}_b)$:
This is the general homomorphic multiplication routine.
1. While $B_a > \lambda \cdot B_{Mult}$ do $\mathfrak{ct}_a \leftarrow BGV.Scale(\mathfrak{ct}_a, \ell_a - 1)$, i.e. keep decreasing the level until the noise bound in $\mathfrak{ct}_a$ is less than $\lambda \cdot B_{Mult}$.
2. While $B_b > \lambda \cdot B_{Mult}$ do $\mathfrak{ct}_b \leftarrow BGV.Scale(\mathfrak{ct}_b, \ell_b - 1)$, i.e. keep decreasing the level until the noise bound in $\mathfrak{ct}_b$ is less than $\lambda \cdot B_{Mult}$.
3. $\ell \leftarrow \min(\ell_a, \ell_b)$.
4. $\mathfrak{ct}_a \leftarrow BGV.Scale(\mathfrak{ct}_a, \ell)$.
5. $\mathfrak{ct}_b \leftarrow BGV.Scale(\mathfrak{ct}_b, \ell)$.
6. Write $\mathfrak{ct}_a = (\mathbf{c}_0^a, \mathbf{c}_1^a, \ell_a, B_a)$.
7. Write $\mathfrak{ct}_b = (\mathbf{c}_0^b, \mathbf{c}_1^b, \ell_b, B_b)$.
8. $\mathbf{d}_0 \leftarrow \mathbf{c}_0^a \odot \mathbf{c}_0^b$.
9. $\mathbf{d}_1 \leftarrow \mathbf{c}_0^a \odot \mathbf{c}_1^b + \mathbf{c}_1^a \odot \mathbf{c}_0^b$.
10. $\mathbf{d}_2 \leftarrow -\mathbf{c}_1^a \odot \mathbf{c}_1^b$.
11. $\mathfrak{ct} \leftarrow BGV.KeySwitch((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_a \cdot B_b)$.
12. $\mathfrak{ct} \leftarrow BGV.Scale(\mathfrak{ct}, \ell - 1)$.
13. Return $\mathfrak{ct}$.

Figure 10: The BGV Leveled Homomorphic Encryption Scheme – Part III.

Notice, that at the end of the *Mult* routine we reduce the level by one, and so this enables us to evaluate circuits of multiplicative depth at most $L - 1$. Recall, we loose one level as we need to consume one level in order to reduce the noise in a fresh ciphertext.

In order to obtain the output of the multiplication operation to be less than B_{Mult} we require, for each ℓ, that

$$\frac{1}{Q_\ell} \cdot \left(\lambda^2 \cdot B_{Mult}^2 + \frac{B_{KeySwitch} \cdot Q^{(\ell)}}{R} + B_{Scale} \right) + B_{Scale} \leq B_{Mult}, \tag{4}$$

where

$$B_{Scale} = c_{err,N} \cdot (P+1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sqrt{N/2})$$

and

$$B_{KeySwitch} = c_{err,N}^2 \cdot P \cdot \sqrt{B/24} \cdot N.$$

This is because the input to the *Scale* operation at the end of the multiplication routine has

$$B_{input} = B_a \cdot B_b \leq \lambda^2 \cdot B_{Mult},$$

since

$$\mathbf{d}_0 - \mathbf{d}_1 \odot \mathfrak{st} - \mathbf{d}_2 \odot \mathfrak{st} \odot \mathfrak{st} = (\mathbf{c}_0^a - \mathbf{c}_1^a \odot \mathfrak{st}) \odot (\mathbf{c}_0^b - \mathbf{c}_1^b \odot \mathfrak{st}).$$

To ensure valid decryption (at any level) we require

$$E_M \cdot B_{Mult} < Q_1/2. \tag{5}$$

So we need to satisfy equations (4) and (5) whilst also maintaining security.

We make the simplifying assumption that all Q_i (except for Q_1) are roughly the same size. We first consider the quadratic equation (4), which we write as

$$\frac{\lambda^2}{Q_\ell} \cdot B_{Mult}^2 - B_{Mult} + \left(\frac{B_{KeySwitch} \cdot Q^{(\ell-1)}}{R} + \frac{B_{Scale}}{Q_\ell} + B_{Scale} \right) \leq 0.$$

Denote the constant term here by C_ℓ, and notice that C_ℓ increases as ℓ increases. We have that $C_\ell > B_{Scale}$, and we would like C_L to be as close to B_{Scale} as possible. To do this we first require that, for all i,

$$Q_i > B_{Scale}.$$

We can obtain $C_L \approx B_{Scale}$ by setting R to be large enough, i.e.

$$
\begin{aligned}
R &\approx \frac{256 \cdot B_{KeySwitch} \cdot Q}{B_{Scale}}, \\
&= \frac{256 \cdot (c_{err,N}^2 \cdot P \cdot \sqrt{B/24} \cdot N) \cdot Q}{c_{err,N} \cdot (P+1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sqrt{N/2})}, \\
&\approx \frac{256 \cdot c_{err,N}^2 \cdot P \cdot \sqrt{B/24} \cdot N \cdot Q}{c_{err,N}^2 \cdot P \cdot N/\sqrt{24}}, \\
&\approx 256 \cdot \sqrt{B} \cdot Q. \tag{6}
\end{aligned}
$$

To have solutions the quadratic equation above must have positive discriminant for all ℓ and so we must have

$$1 - 4 \cdot \frac{\lambda^2}{Q_\ell} \cdot C_\ell > 0.$$

Using the approximation, which we have just established, of $C_\ell \approx B_{Scale}$ this gives us, for all $\ell > 1$,

$$
\begin{aligned}
Q_\ell &\geq 4 \cdot \lambda^2 \cdot B_{Scale} \\
&= 4 \cdot \lambda^2 \cdot c_{err,N} \cdot (P+1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sqrt{N/2}) \\
&\approx 4 \cdot \lambda^2 \cdot c_{err,N}^2 \cdot P \cdot \sqrt{N/12} \cdot \sqrt{N/2} \\
&= 2 \cdot \lambda^2 \cdot c_{err,N}^2 \cdot P \cdot N/\sqrt{6}.
\end{aligned}
$$

So we set

$$Q_\ell = 2.1 \cdot \lambda^2 \cdot c_{err,N}^2 \cdot P \cdot N / \sqrt{6}. \tag{7}$$

As $C_\ell \approx B_{Scale}$ we can ensure that equation (4) is satisfied by setting $B_{Mult} = 2 \cdot B_{Scale}$ since then we will have

$$
\begin{aligned}
\frac{\lambda^2}{Q_\ell} \cdot B_{Mult}^2 - B_{Mult} + \left(\frac{B_{KeySwitch} \cdot Q^{(\ell-1)}}{R} + \frac{B_{Scale}}{Q_\ell} + B_{Scale} \right) &\approx \frac{\lambda^2}{Q_\ell} \cdot B_{Mult}^2 - B_{Mult} + B_{Scale} \\
&\leq \frac{1}{4 \cdot B_{Scale}} \cdot B_{Mult}^2 - B_{Mult} + B_{Scale} \\
&= \frac{1}{4 \cdot B_{Scale}} \cdot (B_{Mult} - 2 \cdot B_{Scale})^2 \\
&= 0.
\end{aligned}
$$

Finally we set

$$Q_1 \approx 3 \cdot E_M \cdot B_{Mult}. \tag{8}$$

Thus we have the following Lemma

Lemma 6. *If equations* (6), (7) *and* (8) *are satisfied, then the BGV scheme can correctly evaluate circuits of depth $L - 1$.*

Note these formulae are slightly different than those in [GHS12] and [HS2O]. The main difference being that our equations depend on N and not $\sqrt{N}$. Recall, the reason for this is due to the way the secret key is generated: In [GHS12] it is chosen to be of fixed Hamming Weight, whilst in [HS2O] it is chosen by rejection sampling until something of small canonical norm is selected. For threshold key generation both of these approaches are relatively complex, thus we require (for BGV) a more simpler secret key generation method, and thus more complex equations.

5.6.1.3 BGV Scheme Security: To ensure security of BGV we need to assume an additional hardness assumption. As the key–switching key contains an LWE sample of the form (for $T = R \cdot Q$)

$$\mathfrak{pf}'_a \leftarrow \mathbf{R}_T, \quad \mathfrak{pf}'_b \leftarrow \mathfrak{pf}'_a \odot \mathfrak{sf} + P \cdot \mathbf{e}'_{\mathfrak{pf}} - R \cdot \mathfrak{sf} \odot \mathfrak{sf} \in \mathbf{R}_T,$$

we actually have that the public key contains an LWE "encryption", under the secret key $\mathfrak{sf}$, of a "message" $R \cdot \mathfrak{sf} \odot \mathfrak{sf}$ which is a function of the secret key. Thus we need to make the circular security assumption that giving out such encryptions does not weaken the security of the underlying LWE problem. We refer to this as the BGV–Circular–Security assumption.

Definition 5 (BGV–Circular–Security Assumption)**.** The Ring–LWE problem given by a sample of the form

$$\mathfrak{pf}_a \leftarrow \mathbf{R}_Q, \quad \mathfrak{pf}_b \leftarrow \mathfrak{pf}_a \odot \mathfrak{sf} + P \cdot \mathbf{e}_{\mathfrak{pf}} \in \mathbf{R}_Q$$

for $\mathbf{e}_{\mathfrak{pf}} \leftarrow NewHope(N, B)$, for a fixed value $\mathfrak{sf} \leftarrow NewHope(N, 1)$, is still hard in the presence of a single sample of the form

$$\mathfrak{pf}'_a \leftarrow \mathbf{R}_T, \quad \mathfrak{pf}'_b \leftarrow \mathfrak{pf}'_a \odot \mathfrak{sf} + P \cdot \mathbf{e}'_{\mathfrak{pf}} - R \cdot \mathfrak{sf} \odot \mathfrak{sf} \in \mathbf{R}_T,$$

where $T = Q \cdot R$ and $\mathbf{e}'_{\mathfrak{pf}} \leftarrow NewHope(N, B)$.

Assuming this we have the following theorem, which is standard

Theorem 1. *The BGV scheme is an IND–CPA Somewhat Homomorphic Encryption scheme assuming the Ring–LWE problem is hard, and the BGV–Circular–Security assumption holds.*

This circular security assumption can be removed if we are willing to have different secret keys at each level of the BGV scheme. This however creates a huge public key which is not very practical.

Then the question arises as to what parameters can we utilize which ensures that the Ring–LWE problem is indeed hard, and we can at the same time evaluate functions homomorphically up to a given level?

We have already derived approximate formulae (in equations (6), (7) and (8)) for R and all the Q_ℓ values, depending only on the parameters $N, c_{err,N}, P, B,$ and L, which ensures correct evaluation of functions up to level L. On top of these equations we need to impose the security constraints in order to derive actual values formulae for Q and R.

To obtain security we need to examine the most insecure Ring–LWE sample we use. This is for the key–switching sample $(\mathfrak{pk}'_a, \mathfrak{pk}'_b)$ in the public key, since all dimensions N are the same in every sample, and the key–switching sample is for the largest ciphertext modulus, i.e. $T = Q \cdot R$. The Ring–LWE sample is derived from a noise distribution of $NewHope(N, B)$, and a secret key distribution of $NewHope(N, 1)$. For a given set of (N, B, T) values, we can then look up the maximum bit-length of T in the Table 23 and determine whether we are in a secure range.

5.6.1.4 Standard BGV Parameter Example:
We present here a simple example parameter size evaluation.] We first select

$$P = 65537, \quad N = 65536, \quad B = 1, \quad c_{err,N} = 14.10, \quad \lambda = 16.$$

To ensure security we assume a maximum bit size of $Q \cdot R$ of 1536; which Table 23 tells us is secure. Choosing $N = 65536$, i.e. a power of two, allows us to take $E_M = 1$.

This allows us to find the following values, from our above equations,

$$B_{Mult} \approx 2^{38.34},$$
$$Q_\ell \approx 2^{47.41} \text{ for } \ell > 1,$$
$$Q_1 \approx 2^{39.92}.$$

Given the bound on $Q \cdot R$ this means we can support at most $L = 16$ levels of multiplications, and we then can derive the following parameter values:

$$Q \approx 2^{751.12},$$
$$R \approx 2^{759.12},$$
$$Q \cdot R \approx 2^{1510.25},$$
$$B_{Scale} \approx 2^{37.34},$$
$$B_{KeySwitch} \approx 2^{37.34},$$
$$B_{ct} \approx 2^{39.63}.$$

5.6.1.5 Enabling Threshold Operations:
To enable all of our threshold algorithms (which for BGV, following Design Decision 14, we only do in the *nSmall* threshold profiles) we need to replace equation (5) with the following constraint

$$2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \cdot E_M \cdot B_{Mult} < Q_1/2.$$

> **Parameter Choice 5: BGV Level 1 Modulus Size**
>
> To enable threshold operations for BGV in threshold profile *nSmall* we set $Q_1 \approx 4 \cdot nSmallBnd \cdot 2^{stat} \cdot E_M \cdot B_{Mult}$.

This is to ensure that we have enough space at the lowest level in order to execute noise flooding. Imposing this constraint in the above analysis results in parameters which have a slightly smaller number of levels. Thus the above constraint turns into the following parameter choice. Recall, following Design Decision 10, we assume a maximum value of $nSmallBnd = 10,000$ in deriving the parameters.

5.6.1.6 Threshold BGV Parameter Example: Imposing this additional constraint on the generation of parameters for BGV we obtain, again starting with

$$P = 65537, \quad N = 65536, \quad B = 1, \quad c_{err,N} = 14.1, \quad \lambda = 16,$$

the parameter sizes

	$P = 65537$
λ	16
N	65536
B	1
λ	16
L	15
B_{Mult}	$\approx 2^{38.34}$
$Q_\ell\ (\ell \neq 1)$	$\approx 2^{47.41}$
Q_1	$\approx 2^{93.63}$
Q	$\approx 2^{757.41}$
R	$\approx 2^{765.41}$
$Q \cdot R$	$\approx 2^{1522.83}$
B_{Scale}	$\approx 2^{37.34}$
$B_{KeySwitch}$	$\approx 2^{37.34}$
$B_{\mathfrak{ct}}$	$\approx 2^{39.63}$

Table 5: Sample parameters for Threshold BGV.

Thus we reduce from being able to evaluate $L = 16$ levels of multiplications to $L = 15$ levels by adding in the capability to perform threshold operations; i.e. enabling threshold operations costs us a single level of homomorphic evaluation.

5.6.2   BFV

In this section we give a brief overview of the BFV [Bra12, FV12] scheme. As for BGV, we do not provide a full optimized specification, as that is beyond the scope of this document. Also, following Design Decision 17, we present a variant of BFV which applies the homomorphic operations by passing directly to BGV–style ciphertexts after the encryption operation.

Thus, due to Design Decision 17, we assume the same restrictions on the ciphertext modulus as in the BGV scheme. That is, a ciphertext modulus in BFV is a product of moduli $Q = Q_1 \cdots Q_L$, each one corresponding (roughly) to a level of multiplications. As we progress through a computation the levels are peeled off, and our ciphertext modulus decreases. We will write $Q = Q^{(L)}$ and $Q^{(\ell)} = Q_1 \cdots Q_\ell$ to be the ciphertext modulus at level ℓ. We start at level L and then decrease levels until we obtain level one. Hence, the parameters associated with our BFV scheme are as follows (P, Q, N, L, R, B), where:

- The plaintext modulus P.
- The top ciphertext modulus $Q = Q_1 \cdots Q_L$.
- The values Q_i are prime with $Q_i \equiv 1 \pmod{2 \cdot N}$.
- The cyclotomic ring $\mathbf{R}^{(M)}$, and its degree $N = \phi(M)$.
- The supported number of levels L.
- A large prime R used for key–switching. We also choose $R \equiv Q_i \equiv 1 \pmod{P}$ to avoid some complications in the methods below, as well as $R \equiv 1 \pmod{2 \cdot N}$.
- A distribution $NewHope(N, B)$ to generate noise values, for a very small value of B (typically $B = 1$). Recall, this is chosen to enable efficient key generation within an MPC engine.

A BFV ciphertext, at level ℓ, will consist of a tuple

$$\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{\mathfrak{ct}})$$

where $\mathbf{c}_0, \mathbf{c}_1 \in \mathbf{R}_{Q^{(\ell)}}$ and $B_{\mathfrak{ct}}$ is a bound on the canonical norm of the pre–decryption of the ciphertext, **when in BGV format**, which holds with probability at least $1 - err$. The ciphertext will encrypt a message $\mathbf{m} \in \mathbf{R}_P$.

The method for converting a BFV format ciphertext into a BGV one, and vice versa, is described in [AP13]. This enables us to define the BFV scheme using BGV homomorphic operations as a sub–routine. We therefore first explain the conversion itself before proceeding to the actual BFV operations.

Recall, a BGV ciphertext $(\mathbf{a}, \mathbf{b}) \in \mathbf{R}_Q^2$ satisfies

$$\mathbf{b} = \mathbf{a} \odot \mathbf{s} + P \cdot \mathbf{e} + \mathbf{m} \pmod{Q}, \tag{9}$$

and a BFV ciphertext $(\mathbf{a}', \mathbf{b}') \in \mathbf{R}_Q^2$ satisfies

$$\mathbf{b}' = \mathbf{a}' \odot \mathbf{s} + \mathbf{e}' + \Delta \cdot \mathbf{m} \pmod{Q}, \tag{10}$$

where we use $\Delta = \lfloor Q/P \rfloor = (Q - 1)/P$. Here we assumed that $Q = 1 \pmod{P}$ as for BGV (more information about the exact format of Q is given later). As specified in Figure 11, conversion from BGV to BFV can be done by multiplication with Δ. Conversely, conversion from BFV to BGV can be done by multiplication with $-P$. The conversion does not change the absolute value of the noise, since we have $\Delta = (Q - 1)/P$.

Scheme conversion

$BGV.toBFV(\mathfrak{ct})$:
   Conversion from BGV to BFV.
   1. Write $\mathfrak{ct} = (\mathbf{a}, \mathbf{b}, \ell, B)$.
   2. $\mathbf{a}' \leftarrow ((Q^{(\ell)} - 1)/P) \cdot \mathbf{a} \pmod{Q^{(\ell)}}$.
   3. $\mathbf{b}' \leftarrow ((Q^{(\ell)} - 1)/P) \cdot \mathbf{b} \pmod{Q^{(\ell)}}$.
   4. $\mathfrak{ct}' \leftarrow (\mathbf{a}', \mathbf{b}', \ell, B)$.
   5. Return $\mathfrak{ct}'$.

$BFV.toBGV(\mathfrak{ct}')$:
   Conversion from BFV to BGV.
   1. Write $\mathfrak{ct}' = (\mathbf{a}', \mathbf{b}', \ell, B)$.
   2. $\mathbf{a} \leftarrow -P \cdot \mathbf{a}' \pmod{Q^{(\ell)}}$.
   3. $\mathbf{b} \leftarrow -P \cdot \mathbf{b}' \pmod{Q^{(\ell)}}$.
   4. $\mathfrak{ct} \leftarrow (\mathbf{a}, \mathbf{b}, \ell, B)$.
   5. Return $\mathfrak{ct}$.

Figure 11: Conversion from BGV to BFV and vice versa.

Lemma 7. *The algorithm $BGV.toBFV(\mathfrak{ct})$ in Figure 11 is correct.*

Proof. To prove that the conversion is correct, and does not affect the size of the noise value, assume that we start from the BGV ciphertext from equation (9). If the input BGV ciphertext is $(\mathbf{a}, \mathbf{b})$, then the resulting converted ciphertext is given by $\mathbf{a}' = \Delta \cdot \mathbf{a} \pmod{Q}$ and $\mathbf{b}' = \Delta \cdot \mathbf{b} \pmod{Q}$. Thus we have

$$
\begin{aligned}
\mathbf{b}' &= \Delta \cdot (\mathbf{a} \odot \mathbf{s} + P \cdot \mathbf{e} + \mathbf{m}) \pmod{Q} \\
&= \mathbf{a}' \odot \mathbf{s} + \Delta \cdot P \cdot \mathbf{e} + \Delta \cdot \mathbf{m} \pmod{Q} \\
&= \mathbf{a}' \odot \mathbf{s} + (Q - 1) \cdot \mathbf{e} + \Delta \cdot \mathbf{m} \pmod{Q} \\
&= \mathbf{a}' \odot \mathbf{s} - \mathbf{e} + \Delta \cdot \mathbf{m} \pmod{Q}.
\end{aligned}
$$

So the noise changes in sign, but does not change in absolute value. □

Lemma 8. *The algorithm $BFV.toBGV(\mathfrak{ct})$ in Figure 11 is correct.*

Proof. Similarly, if we assume that we start from a BFV ciphertext $(\mathbf{a}', \mathbf{b}')$ satisfying equation (10), then the corresponding BGV ciphertext satisfies $\mathbf{a} = -P \cdot \mathbf{a}' \pmod{Q}$ and $\mathbf{b} = -P \cdot \mathbf{b}' \pmod{Q}$. Thus we have

$$
\begin{aligned}
\mathbf{b} &= -P \cdot (\mathbf{a}' \odot \mathbf{s} + \mathbf{e}' + \Delta \cdot \mathbf{m}) \pmod{Q} \\
&= \mathbf{a} \odot \mathbf{s} - P \cdot \mathbf{e}' - P \cdot \Delta \cdot \mathbf{m} \pmod{Q} \\
&= \mathbf{a} \odot \mathbf{s} - P \cdot \mathbf{e}' - (Q - 1) \cdot \mathbf{m} \pmod{Q} \\
&= \mathbf{a} \odot \mathbf{s} - P \cdot \mathbf{e}' + \mathbf{m} \pmod{Q}.
\end{aligned}
$$

Hence, again the noise changes in sign but not in absolute value. □

The main algorithms underlying BFV can now be defined as in Figure 12, with the resulting parameters being identical to those in Table 5. We keep the key–switching key $(\mathfrak{pk}'_a, \mathfrak{pk}'_b)$ in BGV format, since we assume all the homomorphic operations are applied to BGV format ciphertexts.

BFV

BFV.KeyGen(N, Q, P, B, R):
1. $\mathfrak{sk} \leftarrow NewHope(N, 1)$, i.e. each element of the N coefficients are selected from $\{-1, 0, 1\}$ with probabilities $\Pr[0] = 1/2$ and $\Pr[1] = \Pr[-1] = 1/4$.
2. $\mathfrak{pk}_a \leftarrow \mathbf{R}_Q$.
3. $\mathbf{e}_{\mathfrak{pk}} \leftarrow NewHope(N, B)$.
4. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_a \odot \mathfrak{sk} + e_{\mathfrak{pk}} \in \mathbf{R}_Q$.
5. Set $T \leftarrow R \cdot Q$.
6. $\mathfrak{pk}'_a \leftarrow \mathbf{R}_T$.
7. $\mathbf{e}'_{\mathfrak{pk}} \leftarrow NewHope(N, B)$.
8. $\mathfrak{pk}'_b \leftarrow \mathfrak{pk}'_a \odot \mathfrak{sk} + P \cdot \mathbf{e}'_{\mathfrak{pk}} - R \cdot \mathfrak{sk} \odot \mathfrak{sk} \in \mathbf{R}_T$.
9. $\mathfrak{pk} \leftarrow \{(\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}'_a, \mathfrak{pk}'_b)\}$.
10. Return $(\mathfrak{pk}, \mathfrak{sk})$.

BFV.Enc($\mathbf{m}, \mathfrak{pk}$):
Note, the output ciphertext is in BFV format.
1. $\mathbf{v} \leftarrow NewHope(N, 1)$.
2. $\mathbf{e}_0, \mathbf{e}_1 \leftarrow NewHope(N, B)$.
3. $\mathbf{c}_0 \leftarrow \mathfrak{pk}_b \odot \mathbf{v} + \mathbf{e}_0 + \Delta \cdot \mathbf{m} \in \mathbf{R}_Q$.
4. $\mathbf{c}_1 \leftarrow \mathfrak{pk}_a \odot \mathbf{v} + \mathbf{e}_1 \in \mathbf{R}_Q$.
5. $B_{\mathfrak{ct}} \leftarrow c_{err,N} \cdot P \cdot \left(\sqrt{N/12} + \sqrt{B} \cdot (c_{err,N} \cdot N + \sqrt{N/2}) \right)$.
6. Return $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, L, B_{\mathfrak{ct}})$, In some applications we will require the return of $((\mathbf{c}_0, \mathbf{c}_1, L, B_{\mathfrak{ct}}), (\mathbf{v}, \mathbf{e}_0, \mathbf{e}_1))$ in order to prove knowledge and correctness of the encryption via the ZKPoKs in Section 7.6.

BFV.Dec($\mathfrak{ct}, \mathfrak{sk}$):
This assumes the input ciphertext is in BFV format.
1. Write $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{\mathfrak{ct}})$.
2. If $E_M \cdot B_{\mathfrak{ct}} > Q^{(\ell)}/2$ then *abort* (this ciphertext has too much noise).
3. $\mathbf{p} \leftarrow \mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk} \pmod{Q^{(\ell)}}$.
4. If $\mathfrak{ct}$ is in BFV format then $\mathbf{m} \leftarrow \lceil (P/Q^{(\ell)}) \cdot \mathbf{p} \rfloor$.
5. Else $\mathbf{m} \leftarrow \mathbf{p} \pmod{P}$.
6. Return $\mathbf{m}$.

BFV.Add($\mathfrak{ct}_a, \mathfrak{ct}_b$):
Note, the output ciphertext is in BGV format.
1. If $\mathfrak{ct}_a$ is in BFV format then execute $\mathfrak{ct}_a \leftarrow BFV.toBGV(\mathfrak{ct}_a)$.
2. If $\mathfrak{ct}_b$ is in BFV format then execute $\mathfrak{ct}_b \leftarrow BFV.toBGV(\mathfrak{ct}_a)$.
3. $\mathfrak{ct}_c \leftarrow BGV.Add(\mathfrak{ct}_a, \mathfrak{ct}_b)$.
4. Return $\mathfrak{ct}_c$.

BFV.Mult($\mathfrak{ct}_a, \mathfrak{ct}_b$):
Note, the output ciphertext is in BGV format.
1. If $\mathfrak{ct}_a$ is in BFV format then execute $\mathfrak{ct}_a \leftarrow BFV.toBGV(\mathfrak{ct}_a)$.
2. If $\mathfrak{ct}_b$ is in BFV format then execute $\mathfrak{ct}_b \leftarrow BFV.toBGV(\mathfrak{ct}_a)$.
3. $\mathfrak{ct}_c \leftarrow BGV.Mult(\mathfrak{ct}_a, \mathfrak{ct}_b)$.
4. Return $\mathfrak{ct}_c$.

Figure 12: The BFV Leveled Homomorphic Encryption Scheme.

5.6.3 TFHE

In this section we give a brief overview of the TFHE [CGGI16, CGGI20, CLOT21, BdBB$^+$25] scheme. We do not provide a full optimized specification, as that is beyond the scope of this document. We focus instead on what is needed to understand how our threshold schemes work in the context of TFHE. Note, due to the fact that our secret keys are generated inside a threshold protocol we need to deviate from the "classical" representation of the scheme a little. We refer the reader to [CGGI16, CGGI20, CLOT21, BdBB$^+$25] for a full description of the scheme, and the associated proofs of the various formulae presented below.

A ciphertext modulus in TFHE is a power of two $Q = 2^K$, for some value K (usually equal to 64). The plaintext modulus $P = 2^{\mathfrak{p}}$ is also a power of two. Due to our public key encryption methodology, a fresh encryption will actually have LWE dimension $\hat{\ell}$ that will be a power of two; however this will immediately be converted to one whose LWE dimension is either ℓ or $w \cdot N$. This conversion can be performed by the encryptor, or via the service performing the homomorphic operations. However, any zero-knowledge proofs of correct encryption should be applied to the ciphertext of LWE dimension $\hat{\ell}$, i.e. before the key switch operation is performed.

Our TFHE variant has multitudes of secret keys;

- A secret key $\hat{\mathbf{s}} \in \{0, 1\}^{\hat{\ell}}$ which is the underlying secret key associated to the public key encryption method.
- A secret key $\mathbf{s} \in \{0, 1\}^{\ell}$ which is the underlying secret key used for the LWE ciphertexts during the homomorphic operations.
- A set of w secret keys $(\mathbf{s}_0, \ldots, \mathbf{s}_{w-1})$ with $\mathbf{s}_i \in \{0, 1\}^N$ which are used for the GLWE/F-GLWE ciphertexts during the homomorphic operations.
- A set of $\overline{w}$ secret keys $(\overline{\mathbf{s}}_0, \ldots, \overline{\mathbf{s}}_{\overline{w}-1})$ with $\overline{\mathbf{s}}_i \in \{0, 1\}^{\overline{N}}$ which are used to secure the bootstrapping keys used in the *TFHE.SwitchSquash* operation described below.

These secret keys end up being all inter-related via various public key switching and bootstrapping keys. Which keys need to be kept long term in secret shared form depend on the precise method of threshold decryption used, as well as the variant of TFHE which one selects to implement.

Internally, as homomorphic operations are performed, TFHE switches from LWE-style ciphertexts of dimension ℓ, to GLWE-style ciphertexts which consists of a Ring-LWE dimension N (again a power of two) plus a "repetition factor"/"GLWE-dimension" w, to so-called flattened (or F-GLWE) ciphertexts, which are essentially LWE-style ciphertext of dimension $w \cdot N$. In all cases we will be encrypting a message $m \in \mathbb{Z}/(P)$. An implementer needs to decide whether to use type *LWE* or *F-GLWE* as the representation of ciphertexts for performing homomorphic operations, via a global variable which will be denoted *type* below. The global *type* value chosen impacts on how one derives the noise formulae, and hence parameters, where as *ctType* will denote the type of a specific ciphertext in the ensuing algorithms.

- For *ctType = LWE* a ciphertext consists of a tuple

$$\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1)$$

where $\mathbf{c}_0 \in (\mathbb{Z}/(Q))^{\ell}$ and $c_1 \in \mathbb{Z}/(Q)$. We shall refer to $p = \mathbf{c}_1 - \mathbf{c}_0 \cdot \mathbf{s} \in \mathbb{Z}/(Q)$ as the pre-decryption value of this ciphertext.

- For *ctType = GLWE* a ciphertext consists of a tuple

$$\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1)$$

where $\mathbf{c}_0 = (\mathbf{\alpha}_0, \ldots, \mathbf{\alpha}_{w-1}) \in \mathbf{R}_Q^{\ell}$ and $c_1 \in \mathbf{R}_Q$. Here we shall refer to $\mathbf{p} = \mathbf{\alpha}_1 - \sum_{i=0}^{w-1} \mathbf{\alpha}_i \odot \mathbf{s}_i \in \mathbf{R}_Q$ as the pre-decryption value of this ciphertext. A GLWE ciphertext can encrypt an element $\mathbf{m} \in \mathbf{R}_P$. GLWE ciphertexts are not exposed externally to routines, but internally within the bootstrapping

routines we utilize GLWE ciphertexts to process encryptions of elements in $\mathbf{R}_P$.

· For $ctType = F\text{-}GLWE$ a ciphertext consists of a tuple

$$\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1)$$

where $\mathbf{c}_0 = ((\mathbf{a}_{i,j})_{j=0}^{N-1})_{i=0}^{w-1} \in \mathbb{Z}/(Q)^{w \times N}$ and $c_1 \in \mathbb{Z}/(Q)$. Here the pre–decryption value is $p = \mathbf{a}_1 - \sum_{i=0}^{w-1} \sum_{j=0}^{N-1} \mathbf{a}_{i,j} \cdot \mathbf{s}_i[j] \in \mathbb{Z}/(Q)$.

We will denote the ciphertext type of a given ciphertext $\mathfrak{ct}$ by the function $ctType(\mathfrak{ct})$, which will be an element of the set $\{LWE, GLWE, F\text{-}GLWE\}$, although, as mentioned above, in practice we do not store ciphertexts of type $GLWE$ long term. Generally speaking setting $type = F\text{-}GLWE$ results in a more efficient scheme, as was demonstrated in [CJP21, BBB+23].

Note we (unlike for BGV and BFV) do not keep an estimate for the noise with the ciphertext. This is because the noise bounds do not depend on the topology of the underlying computation being per–formed, since all computations are performed in a relatively regular manner of linear operations followed by a programmable bootstrap. However, for all our operations defined below on TFHE ciphertexts we will need to keep track of how much each operation affects the mean and standard deviation of either $|p - \Delta \cdot m|$ or $\|\mathbf{p} - \Delta \cdot \mathbf{m}\|_\infty$. We will then combine these various estimates together in order to derive equations which guarantee correctness and security, and from these we will derive parameters.

Given a ciphertext encrypting $\mathbf{m} \in \mathbf{R}_P$ of type $GLWE$, we can transform it into a ciphertext of type $F\text{-}GLWE$ using a process called flattening, also known in the literature as the sample extract operation. Flattening simply involves reordering of the coefficients and negating some of them as described in Figure 13. Note that, in general, the flattening operation, goes from a GLWE ciphertext encrypting a polynomial $\mathbf{m} \in \mathbf{R}_P$, to an F–GLWE ciphertext encrypting only the constant term of that polynomial, i.e. an element $m \in \mathbb{Z}/(P)$. Thus the flattening operation, in general, is not invertible. The flattening operation does not alter any statistical properties of the value of $|p - \Delta \cdot m|$, it is purely a format conversion operation.

TFHE Flattening

The input ciphertext $\mathfrak{ct}$ is a ciphertext of type $GLWE$ and the output ciphertext of type $F\text{-}GLWE$. If the input ciphertext is an encryption of $\mathbf{m}$, the output is an encryption of $\mathbf{m}[0]$.

$TFHE.Flatten(\mathfrak{ct}; N, w)$:
  1. Write $\mathfrak{ct} = ((\mathbf{a}_0, \ldots, \mathbf{a}_{w-1}), \mathbf{b})$.
  2. $\mathbf{a}' \leftarrow (\mathbf{a}_0[0], -\mathbf{a}_0[N-1], \ldots, -\mathbf{a}_0[1], \ldots, \mathbf{a}_{w-1}[0], -\mathbf{a}_{w-1}[N-1], \ldots, -\mathbf{a}_{w-1}[1])$.
  3. $b' \leftarrow \mathbf{b}[0]$.
  4. Return $(\mathbf{a}', b')$.

Figure 13: TFHE Flattening Algorithm from GLWE to F–GLWE (also called sample extract).

To simplify analysis we do not consider the following aspects of TFHE.
  1. We do not consider any "structure" in the plaintext space (for example padding bits etc), as we leave that as an issue for an application layer. This implies that any function passed into the programmable bootstrapping operation (PBS) below, needs to be negacyclic on the encrypted input range (we will discuss this in further detail below).
  2. The modulus switch algorithm, in Figure 24, outputs a ciphertext whose noise is from a distribution which is not centered around zero; it has a very small expected value, due to the rounding in the algorithm. In our analysis we neglect the tiny change in the distributions that this induces. A more careful analysis can remove this by using a probabilistic rounding methodology.

The above optimizations/changes can be added as required, with little (if any) changes to our threshold protocols.

We make an implicit assumption that the noise distributions on the output of the bootstrapping and key switching procedures are distributed roughly like a Gaussian with the given standard deviation;

experimentally this seems to be true.

A key aspect of TFHE is the fact that one can perform bootstrapping operations efficiently. In addition one is able, during a bootstrapping operation, to evaluate an arbitrary table–look–up operation, a so called programmable bootstrapping (PBS). To enable this we need to define various other parameters. As well as P, Q, $\hat{\ell}$, ℓ, w and N, there are additional parameters (β_{pksk}, ν_{pksk}, β_{ksk}, ν_{ksk}, β_{bk}, ν_{bk}) which correspond to the three key–switching keys and the bootstrapping keys decomposition bases (β_{pksk}, β_{ksk} and β_{bk}), as well as the key–switching keys and bootstrapping keys decomposition levels (ν_{pksk}, ν_{ksk} and ν_{bk}).

One of our methodologies to perform threshold decryption requires us to change the main parameters (ℓ, Q) underlying a given ciphertext $\mathfrak{ct}$, to a larger set ($\overline{\ell}$, $\overline{Q}$). In doing so we obtain a new ciphertext $\overline{\mathfrak{ct}}$ which encrypts the same message as $\mathfrak{ct}$, but with these larger switched parameters. In addition the "noise" underlying $\overline{\mathfrak{ct}}$ has been squashed. We refer to this operation, of passing from $\mathfrak{ct}$ to $\overline{\mathfrak{ct}}$ as *TFHE.SwitchSquash*. To implement *TFHE.SwitchSquash*, see Figure 28 later, we perform a bootstrapping operation with different parameters. These we denote by

$$\overline{N}, \overline{w}, \overline{\beta_{bk}}, \overline{\nu_{bk}}$$

and we set $\overline{\ell} = \overline{w} \cdot \overline{N}$. To indicate whether the extra data for this form of threshold decryption needs to be produced we use a flag $\overline{flag}$, which is true when the extra data is required.

5.6.3.1   The TFHE Scheme Description: To summarize the (basic) parameters associated with our TFHE scheme are thus as follows

$$(P, Q, type, \ell, b_\ell, (\hat{\ell}, b_{\hat{\ell}}, \beta_{pksk}, \nu_{pksk}), (N, w, b_{w \cdot N}, \beta_{ksk}, \nu_{ksk}, \beta_{bk}, \nu_{bk})),$$

and when using *TFHE.SwitchSquash* to these we need to add the parameters

$$(\overline{N}, \overline{w}, b_{\overline{w} \cdot \overline{N}}, \overline{\beta_{bk}}, \overline{\nu_{bk}}),$$

where we have
- The plaintext modulus P (a power of two).
- The ciphertext modulus Q (a power of two).
- Whether we apply homomorphic operations to ciphertexts in $type = LWE$ or $type = F\text{-}GLWE$ format.
- Two cyclotomic rings $\mathbf{R}^{(M)}$ and $\mathbf{R}^{(2 \cdot \hat{\ell})}$, with degrees $N = \phi(M)$ and $\hat{\ell} = \phi(2 \cdot \hat{\ell})$, (both N and $\hat{\ell}$ are powers of two).
- The "repetition factor" w for the Generalized–LWE problem.
- The main LWE dimension ℓ used during homomorphic computations.
- A set of parameters ($\hat{\ell}$, $b_{\hat{\ell}}$, β_{pksk}, ν_{pksk}) associated with public key encryption, via the method of [Joy24] to create a ciphertext of LWE dimension $\hat{\ell}$ (as above), followed by the conversion to an LWE ciphertext of dimension ℓ. These parameters are used to define the different rings and noise distributions required to define the keys for the conversion of ciphertexts encrypted with the public key to ones which can be computed on homomorphically.
- A set of parameters associated with the PBS operation (N, w, $b_{w \cdot N}$, β_{ksk}, ν_{ksk}, β_{bk}, ν_{bk}). These parameters are used to define the different rings and noise distributions required to define the keys for the bootstrapping and key switching operations that we require (N is the same value as above).
- We will need to define three/four noise distributions, one to work with the LWE dimension ℓ, which again will be chosen to be of the form $TUniform(\cdot, -2^{b_\ell}, 2^{b_\ell})$. A second to work with the LWE dimension $\hat{\ell}$, again this will be chosen to be of the form $TUniform(\cdot, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$. Another to generate the noise distributions for the GLWE problem underlying the PBS operation, for this we also use a noise distribution of the form $TUniform(\cdot, -2^{b_{w \cdot N}}, 2^{b_{w \cdot N}})$. And potentially a fourth to deal

with the *TFHE.SwitchSquash* GLWE problem, for which we use the noise distribution of the form $TUniform(\cdot, -2^{b_{\overline{w}\cdot\overline{N}}}, 2^{b_{\overline{w}\cdot\overline{N}}})$. As before, these distributions are chosen to enable efficient threshold key generation within an MPC engine.

Many of these points follow from our Design Decision 19. Note the values $b_\star$ for various values of $\star$ we will think of as being defined by the subscript $\star$. So we will not pass them as parameters to functions, but consider them as derived from the value $\star$ (in addition to the desired security parameter *sec*).

5.6.3.2 Public Key Compression via a XOF:
As one can see the number of parameters is quite large, and the resulting public keys/evaluation keys for TFHE will also be quite large. This is because we need to store both the public key encryption keys and the keys for the PBS operations. In order to reduce storage of public keys we use a form of public key compression via a XOF, see Section 5.1.

Use of a XOF allows us to compress the random elements in the public key into a single XOF key, and then expand them when required by calling the XOF. As long as the XOF is called in the correct order for the correct variables one will recover the correct values for the public key. This is enabled via the functions *Expand* below.

5.6.3.3 TFHE Public Key Generation:
We are now in a position to define the TFHE public key. This is built up in a sequence of steps in Figure 14–Figure 18. Notice, each encryption algorithm can either return the components **α** and b, or just the component b. This depends on whether the calling algorithm requires the expanded version of **α** or not; this is signaled via the flag *flag*.

The Basic TFHE Operations: Key Gen Part I

This defines the basic (internal) TFHE symmetric key encryption operations; the encryption functions here *are not* the externally exposed public key ones; they are used to define key switching material, etc.

$Enc^{LWE}(m, \mathbf{s}; XOF, P', Q, \ell, flag)$:
  For $m \in \mathbb{Z}/(P')$ and $\mathbf{s} \in \{0, 1\}^\ell$ this produces an LWE ciphertext. Note, the plaintext modulus P' can vary here; but it will always be a power of two, and less than Q.
  1. $\mathbf{\alpha} \leftarrow Expand^{LWE}(XOF, Q, \ell)$.
  2. $e \leftarrow TUniform(1, -2^{b_\ell}, 2^{b_\ell})$.
  3. $b \leftarrow \mathbf{\alpha} \cdot \mathbf{s} + e + (Q/P') \cdot m \pmod{Q}$.
  4. If $flag$ return $(\mathbf{\alpha}, b)$, else return b.

$Enc^{GLWE}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); XOF, P', Q, N, w, flag)$:
  For **R** the ring of dimension N and $\mathbf{m} \in \mathbf{R}_{P'}, \mathbf{s}_i \in \mathbf{R}_Q$ this produces a GLWE ciphertext. The same note around P' being a power of two, less than Q applies here as well.
  1. $(\mathbf{\alpha}_0, \dots, \mathbf{\alpha}_{w-1}) \leftarrow Expand^{GLWE}(XOF, Q, N, w)$.
  2. $\mathbf{e} \leftarrow TUniform(N, -2^{b_{w\cdot N}}, 2^{b_{w\cdot N}})$.
  3. $\mathbf{b} \leftarrow \sum_{i=0}^{w-1} \mathbf{\alpha}_i \odot \mathbf{s}_i + \mathbf{e} + (Q/P') \cdot \mathbf{m} \pmod{Q}$.
  4. If $flag$ return $(\mathbf{\alpha}_0, \dots, \mathbf{\alpha}_{w-1}, \mathbf{b})$, else return $\mathbf{b}$.

Figure 14: The Basic TFHE Operations: key Gen Part I.

The Basic TFHE Operations: Part II

$Enc^{Lev}(m, \mathbf{s}; XOF, \beta, Q, \ell, \nu, flag)$:

This will only ever be used to encrypt messages $m \in \{0, 1\}$, thus when encrypting we treat m as an element in $\mathbb{Z}/(\beta^{i+1})$ in the application of Enc^{LWE} below. We also always have $\mathbf{s} \in \{0, 1\}^{\ell}$.

1. For $i \in [0, \ldots, \nu - 1]$ do
   (a) $\mathfrak{ct}_i \leftarrow Enc^{LWE}(m, \mathbf{s}; XOF, \beta^{i+1}, Q, \ell, flag)$.
2. Return $(\mathfrak{ct}_0, \ldots, \mathfrak{ct}_{\nu-1})$.

$Enc^{GLev}(\mathbf{m}, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); XOF, \beta, Q, N, w, \nu, flag)$:

Here we assume β is a power of two, and that $\mathbf{m}$ is a message which lies in $\mathbf{R}_{\beta^{\nu}}$, where $\mathbf{R}$ is the ring of dimension N.

1. For $i \in [0, \ldots, \nu - 1]$ do
   (a) $\mathfrak{ct}_i \leftarrow Enc^{GLWE}(\mathbf{m} \pmod{\beta^{i+1}}, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); XOF, \beta^{i+1}, Q, N, w, flag)$.
   In practice the reduction modulo β^{i+1} is not needed as the final reduction modulo Q will perform the same effect. We include it here purely to match up the semantics of the plaintext spaces.
2. Return $(\mathfrak{ct}_0, \ldots, \mathfrak{ct}_{\nu-1})$.

$Enc^{GGSW}(m, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); XOF, Q, N, w, \beta, \nu, flag)$:

Here we assume β is a power of two, and that m is a message which lies in $\{0, 1\}$.

1. For $i \in [0, \ldots, w - 1]$ do
   (a) $\mathfrak{ct}_i \leftarrow Enc^{GLev}(-\mathbf{s}_i \cdot m \pmod{\beta^{\nu}}, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); XOF, \beta, Q, N, w, \nu, flag)$.
   Again, in practice the reduction modulo β^{ν} is not needed, we only include it for matching plaintext space semantics.
2. $\mathfrak{ct}_w \leftarrow Enc^{GLev}(m, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); XOF, \beta, Q, N, w, \nu, flag)$.
   The single bit message m here is lifted to the constant polynomial in the ring $\mathbf{R}_{\beta^{\nu}}$.
3. Return $(\mathfrak{ct}_0, \ldots, \mathfrak{ct}_w)$.

Figure 15: The Basic TFHE Operations: Part II.

$Expand^{LWE}(XOF, Q, \ell)$:
      1. $\boldsymbol{\alpha} \leftarrow XOF.Next(\ell, Q)$.
      2. Return $\boldsymbol{\alpha}$.

$Expand^{GLWE}(XOF, Q, N, w)$:
      1. For $i \in [0, \dots, w-1]$ do
         (a) $\boldsymbol{\alpha}_i \leftarrow XOF.Next(N, Q)$.
      2. Return $(\boldsymbol{\alpha}_0, \dots, \boldsymbol{\alpha}_{w-1})$.

$Expand^{Lev}(XOF, Q, \ell, \nu)$:
      1. For $i \in [0, \dots, \nu-1]$ do
         (a) $\boldsymbol{\alpha}_i \leftarrow Expand^{LWE}(XOF, Q, \ell)$.
      2. Return $(\boldsymbol{\alpha}_0, \dots, \boldsymbol{\alpha}_{\nu-1})$.

$Expand^{GLev}(XOF, Q, N, w, \nu)$:
      Note $\boldsymbol{\alpha}_i$ here is a vector of ring elements of dimension w, thus we return in effect a $\nu \times w$ matrix of ring elements.
      1. For $i \in [0, \dots, \nu-1]$ do
         (a) $\boldsymbol{\alpha}_i \leftarrow Expand^{GLWE}(XOF, Q, N, w)$.
      2. Return $\mathbf{A} = (\boldsymbol{\alpha}_0, \dots, \boldsymbol{\alpha}_{\nu-1})$.

$Expand^{GGSW}(XOF, Q, N, w, \nu)$:
      1. For $i \in [0, \dots, w]$ do
         (a) $\mathbf{A}_i \leftarrow Expand^{GLev}(XOF, Q, N, w, \nu)$.
      2. Return $(\mathbf{A}_0, \dots, \mathbf{A}_w)$.

Figure 16: The Basic TFHE Operations: Part III.

The Public Key TFHE Scheme: Key Gen I

This defines the actual public key version of the scheme.

$TFHE.KeyGen(P, Q, type, \hat{\ell}, \ell, N, w, \beta_{pksk}, \beta_{ksk}, \beta_{bk}, \nu_{pksk}, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \overline{Q}, \overline{\beta_{bk}}, \overline{\nu_{bk}}, flag, \overline{flag})$:
1. $seed \leftarrow \{0, 1\}^{sec}$.
2. $XOF.Init(seed, DSep(TFHE))$.
3. $\hat{\mathbf{s}} \leftarrow \{0, 1\}^{\hat{\ell}}$.
4. $\mathbf{s} \leftarrow \{0, 1\}^{\ell}$.
5. For $i \in [0, \dots, w - 1]$ do $\mathbf{s}_i \leftarrow \{0, 1\}^N$.
6. $\mathfrak{pk}_a \leftarrow XOF.Next(\hat{\ell}, Q)$.
7. $\mathbf{e} \leftarrow TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$.
8. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_a \odot \overleftrightarrow{\hat{\mathbf{s}}} + \mathbf{e}$.
9. For $i \in [0, \dots, w - 1]$ do
   (a) For $j \in [0, \dots, N - 1]$ do $KSK_{i,j} \leftarrow Enc^{Lev}(\mathbf{s}_i[j], \mathbf{s}; XOF, \beta_{ksk}, Q, \ell, \nu_{ksk}, flag)$.
10. If $\overline{flag}$ then
   (a) For $i \in [0, \dots, \overline{w} - 1]$ do $\overline{\mathbf{s}}_i \leftarrow \{0, 1\}^{\overline{N}}$.
11. For $i \in [0, \dots, \ell - 1]$ do
   (a) $BK_i \leftarrow Enc^{GGSW}(\mathbf{s}[i], (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); XOF, Q, N, w, \beta_{bk}, \nu_{bk}, flag)$.
   (b) If $\overline{flag}$ then $\overline{BK}_i \leftarrow Enc^{GGSW}(\mathbf{s}[i], (\overline{\mathbf{s}}_0, \dots, \overline{\mathbf{s}}_{\overline{w}-1}); XOF, \overline{Q}, \overline{N}, \overline{w}, \overline{\beta_{bk}}, \overline{\nu_{bk}}, flag)$.
   (c) Else $\overline{BK}_i \leftarrow \perp$.
12. If $type = F\text{-}GLWE$ then $\mathbf{s} \leftarrow (\mathbf{s}_0[0], \dots, \mathbf{s}_0[N - 1], \mathbf{s}_1[0], \dots, \mathbf{s}_1[N - 1], \dots, \mathbf{s}_{w-1}[0], \dots, \mathbf{s}_{w-1}[N - 1])$.
13. For $i \in [0, \dots, \hat{\ell} - 1]$ do
   (a) If $type = LWE$ then $PKSK_i \leftarrow Enc^{Lev}(\hat{\mathbf{s}}[i], \mathbf{s}; XOF, \beta_{pksk}, Q, \ell, \nu_{pksk}, flag)$.
   (b) Else $PKSK_i \leftarrow Enc^{Lev}(\hat{\mathbf{s}}[i], \mathbf{s}; XOF, \beta_{pksk}, Q, w \cdot N, \nu_{pksk}, flag)$.
14. If $flag$ then
   (a) $\mathfrak{pk} \leftarrow (\mathfrak{pk}_a, \mathfrak{pk}_b)$.
   (b) $PK \leftarrow (\mathfrak{pk}, \{PKSK_i\}_i, \{KSK_{i,j}\}_{i,j}, \{BK_i\}_i, \{\overline{BK}_i\}_i)$.
15. Else
   (a) $PK_b \leftarrow (\mathfrak{pk}_b, \{PKSK_i\}_i, \{KSK_{i,j}\}_{i,j}, \{BK_i\}_i, \{\overline{BK}_i\}_i)$.
   (b) $PK \leftarrow (seed, PK_b)$.
16. If $\overline{flag}$ then $\mathfrak{sk} \leftarrow (\mathbf{s}, (\overline{\mathbf{s}}_0, \dots, \overline{\mathbf{s}}_{\overline{w}-1}))$, else $\mathfrak{sk} \leftarrow (\mathbf{s}, \perp)$.
17. If FFT optimizations are to be applied, then the data in BK could be translated into the Fourier domain.
18. Return $(PK, \mathfrak{sk})$.

Figure 17: The Public Key TFHE Scheme: Key Gen I.

The Public Key TFHE Scheme: Key Gen II

$TFHE.Expand(seed; Q, type, \ell, N, w, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \overline{Q}, \overline{\nu_{bk}}, \overline{flag})$:

    This computes the a–component for every associated element in the public key.

      1. $XOF.Init(seed, DSep(TFHE))$.

      2. $\mathfrak{pk}_a \leftarrow XOF.Next(\hat{\ell}, Q)$.

      3. For $i \in [0, \ldots, w-1]$ do

        (a) For $j \in [0, \ldots, N-1]$ do

            i. $KSK_{i,j}^a \leftarrow Expand^{Lev}(XOF, Q, \ell, \nu_{ksk})$.

      4. For $i \in [0, \ldots, \ell-1]$ do

        (a) $BK_i^a \leftarrow Expand^{GGSW}(XOF, Q, N, w, \nu_{bk})$.

        (b) If $\overline{flag}$ then $\overline{BK}_i^a \leftarrow Expand^{GGSW}(XOF, \overline{Q}, \overline{N}, \overline{w}, \overline{\nu_{bk}})$.

        (c) Else $\overline{BK}_i^a \leftarrow \perp$.

      5. For $i \in [0, \ldots, \hat{\ell}-1]$ do

        (a) If $type = LWE$ then $PKSK_i^a \leftarrow Expand^{Lev}(XOF, Q, \ell, \nu_{pksk})$.

        (b) Else $PKSK_i^a \leftarrow Expand^{Lev}(XOF, Q, w \cdot N, \nu_{pksk})$.

      6. $PK_a \leftarrow (\mathfrak{pk}_a, \{PKSK_i^a\}_i, \{KSK_{i,j}^a\}_{i,j}, \{BK_i^a\}_i, \{\overline{BK}_i^a\}_i)$.

      7. Return PK_a.

Figure 18: The Public Key TFHE Scheme: Key Gen II.

5.6.3.4 Dimension Switching: Due to the restriction that $\hat{\ell}$ has to be a power of two, before the server actually performs any homomorphic operations on a freshly encrypted LWE ciphertext, the server[13] will first convert the freshly encrypted ciphertext to a ciphertext of type *type*. Recall, if *type = LWE* this is an LWE ciphertext of dimension ℓ, and if *type = F-GLWE* this is an F-GLWE ciphertext of dimension $w \cdot N$, which is actually an LWE ciphertext of dimension $w \cdot N$.

The reason for operating with a different value ℓ or $w \cdot N$, as opposed to $\hat{\ell}$, is that ℓ and $w \cdot N$ will be chosen to minimize the cost of performing other operations; in particular the programmable bootstrapping which enables homomorphic operations. In the case of *type = LWE* this means that ℓ can be smaller than $\hat{\ell}$, leading to more efficient homomorphic operations. In the case of *type = F-GLWE* this means that $\hat{\ell}$ can be smaller than $w \cdot N$, leading to more efficient zero–knowledge proofs.

We call this operation a DimensionSwitch; it takes an LWE ciphertext of dimension $\hat{\ell}$ and returns an LWE ciphertext of dimension ℓ or $w \cdot N$, encrypting the same message. The algorithm for DimensionSwitch, given in Figure 20, makes use of a sub–procedure for integer decomposition, which is given in Figure 19.

TFHE Decomposition

TFHE.Decompose(x, Q, β, ν) :

The input x is an integer in $\mathbb{Z}_Q$, the output is a vector **x** of length ν, with $\mathbf{x}[i] \in [-\beta/2, \beta/2]$ and $\sum_{i=0}^{\nu-1} \mathbf{x}[i] \cdot \beta^{\nu-i-1} \equiv \lceil x \cdot \beta^\nu / Q \rfloor \bmod \beta^\nu$. This is based on [Joy21, Algorithm 3]. It uses an internal boolean value $flag_{Decompose}$ which is retained between calls, and which is initially set to zero.

1. Consider x as an integer (i.e. in $\mathbb{Z}$ not $\mathbb{Z}/(Q)$).
2. $x' \leftarrow \lceil x \cdot \beta^\nu / Q \rfloor$.
3. If $x' > \beta^\nu/2$ then $x' \leftarrow x' - \beta^\nu$.
4. If $x' = \beta^\nu/2$ then
   (a) If $flag_{Decompose} = 1$ then do $x' \leftarrow x - \beta^\nu$.
   (b) $flag_{Decompose} \leftarrow 1 - flag_{Decompose}$.
5. For $i \in [0, \ldots, \nu - 1]$ do:
   (a) $x_i \leftarrow x' \bmod \beta$.
   (b) $x' \leftarrow (x' - x_i)/\beta$.
   (c) If $x_i > \beta/2$ or ($x_i = \beta/2$ and $x' \bmod \beta \geq \beta/2$) then
       i. $x_i \leftarrow x_i - \beta$.
       ii. $x' \leftarrow x + 1$.
6. Return $\mathbf{x} = (x_{\nu-1}, x_{\nu-2}, \ldots, x_0)$.

Figure 19: TFHE Decomposition Algorithm mapping an integer modulo Q to its decomposition of length ν and base β.

The *TFHE.DimensionSwitch* operation does not alter the mean of the value $|p - \Delta \cdot m|$, however it does increase the variance by the value σ_{DS}^2 below. For a full derivation of this fact, and other such derivations in this document, see [CLOT21].

$$\sigma_{DS}^2 = \hat{\ell} \cdot \left(\frac{Q^2}{12 \cdot \beta_{pksk}^{2 \cdot \nu_{pksk}}} - \frac{1}{12} \right) \cdot (Var(s_i) + Exp^2(s_i))$$

$$+ \frac{\hat{\ell}}{4} \cdot Var(s_i) + \hat{\ell} \cdot \nu_{pksk} \cdot \sigma_{pksk}^2 \cdot \left(\frac{\beta_{pksk}^2 + 2}{12} \right)$$

$$= \frac{\hat{\ell}}{2} \cdot \left(\frac{Q^2}{12 \cdot \beta_{pksk}^{2 \cdot \nu_{pksk}}} - \frac{1}{12} \right)$$

$$+ \hat{\ell} \cdot \left(\frac{1}{16} + \nu_{pksk} \cdot \sigma_{pksk}^2 \cdot \left(\frac{\beta_{pksk}^2 + 2}{12} \right) \right)$$

---

[13]This could also be carried out by the encryptor if desired.

$$= \hat{\ell} \cdot \left(\frac{Q^2}{24 \cdot \beta_{pksk}^{2 \cdot \nu_{pksk}}} + \frac{1}{48} + \nu_{pksk} \cdot \sigma_{pksk}^2 \cdot \left(\frac{\beta_{pksk}^2 + 2}{12} \right) \right) \tag{11}$$

since, for a binary secret key, we have $Var(s_i) = 1/4$ and $Exp(s_i) = 1/2$, where

$$\sigma_{pksk} = \begin{cases} \sigma_{b_\ell} = \sqrt{(2^{2 \cdot b_\ell + 1} + 1)/6} & \text{if } type = LWE, \\ \sigma_{b_{w \cdot N}} = \sqrt{(2^{2 \cdot b_{w \cdot N} + 1} + 1)/6} & \text{if } type = F\text{-}GLWE. \end{cases}$$

TFHE Dimension Switching

$TFHE.DimensionSwitch(\mathfrak{ct}, PKSK; Q, d, \hat{\ell}, \beta_{pksk}, \nu_{pksk})$:

The input ciphertext is an LWE ciphertext of dimension $\hat{\ell}$, the output ciphertext is an LWE ciphertext of dimension d encrypting the same message.

1. Write $\mathfrak{ct} = (\mathbf{a}, b)$.
2. Parse $PKSK$ as $\{PKSK_i\}_i$ for $i \in [0, \ldots, \hat{\ell} - 1]$.
3. For $i \in [0, \ldots, \hat{\ell} - 1]$ do:
   (a) $\mathbf{d}_i \leftarrow TFHE.Decompose(\mathbf{a}[i], Q, \beta_{pksk}, \nu_{pksk})$.
4. For $i \in [0, \ldots, \hat{\ell} - 1]$ do:
   (a) Parse $PKSK_i$ as $(PKSK_{i,0}, \ldots, PKSK_{i,\nu_{pksk}-1})$ and parse $PKSK_{i,k}$ as $(PKSK_{i,k}^a, PKSK_{i,k}^b)$ for each $k \in [0, \ldots, \nu_{pksk} - 1]$.
   (b) Define the ν_{pksk}–by–d matrix

   $$M_i \leftarrow \begin{pmatrix} PKSK_{i,0}^a[0] & \ldots & PKSK_{i,0}^a[d-1] \\ \vdots & & \vdots \\ PKSK_{i,\nu_{pksk}-1}^a[0] & \ldots & PKSK_{i,\nu_{pksk}-1}^a[d-1] \end{pmatrix},$$

   where $PKSK_{i,k}^a[t]$ is the t–th element of the vector $PKSK_{i,k}^a$.
   (c) $\mathbf{v}_i \leftarrow (PKSK_{i,0}^b, \ldots, PKSK_{i,\nu_{pksk}-1}^b)$, i.e. a ν_{pksk}–dimensional vector.
5. $\mathbf{a}' \leftarrow -\sum_{i=0}^{\hat{\ell}-1} \mathbf{d}_i \cdot M_i$.
   Note this is a sum of $\hat{\ell}$ vector by matrix products.
6. $b' \leftarrow b - \sum_{i=0}^{\hat{\ell}-1} \mathbf{d}_i \cdot \mathbf{v}_i$.
   Note this is a sum of $\hat{\ell}$ vector by vector inner products.
7. Return $(\mathbf{a}', b')$.

Figure 20: TFHE Dimension Switching Algorithm mapping an LWE ciphertext of dimension $\hat{\ell}$ to an LWE ciphertext of dimension d..

5.6.3.5   TFHE Public Key Encryption and Decryption: In this section we present a method for public key encryption, described in Joye [Joy24], which uses smaller public keys. The method of Joye goes via a Ring–LWE encryption, as opposed to the multiple encryptions of zero methodology; the latter being the method which is more commonly used in the literature. The method works for two–power values of $\hat{\ell}$, i.e. we have $\hat{\ell} = 2^\gamma$ for some integer γ. The public key encryption algorithm makes use of the cyclotomic ring $\mathbf{R}^{(2 \cdot \hat{\ell})}$ of dimension $\hat{\ell}$. Thus for TFHE there are at most three cyclotomic rings, one consisting of polynomials of degree at most $\hat{\ell} - 1$ and one consisting of polynomials of degree at most $N - 1$, and (potentially) one consisting of polynomials of degree at most $\overline{N} - 1$. Which one is being considered at any given point should be clear from the context and/or dimension of the elements.

The public key is created by first generating a uniform binary secret key $\hat{\mathbf{s}} \leftarrow \{0, 1\}^{\hat{\ell}}$, then selecting

$\mathfrak{pk}_a \in (\mathbb{Z}/(Q))^{\hat{\ell}}$ uniformly at random and finally generating $\mathfrak{pk}_b \in (\mathbb{Z}/(Q))^{\hat{\ell}}$ via

$$\mathfrak{pk}_b \leftarrow \mathfrak{pk}_a \odot \overleftrightarrow{\hat{\mathbf{s}}} + \mathbf{e}$$

where $\mathbf{e} \leftarrow TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$. Recall, here $\overleftrightarrow{\hat{\mathbf{s}}}$ is the reverse ordering of $\hat{\mathbf{s}}$.

The Public Key TFHE Scheme: Encryption and Decryption

TFHE.Enc-Sub$(m, \mathfrak{pk}; P, Q, \hat{\ell})$:
  Here $\mathfrak{pk}$ is the public key $(\mathfrak{pk}_a, \mathfrak{pk}_b,)$ generated in the *TFHE.KeyGen* algorithm above. The parameters P, Q and $\hat{\ell}$ may be dropped if they can be implicitly inferred from the context. The output is an LWE ciphertext of dimension $\hat{\ell}$.
  1. $\mathbf{r} \leftarrow \{0, 1\}^{\hat{\ell}}$.
  2. $\mathbf{e}_1 \leftarrow TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$.
  3. $e_2 \leftarrow TUniform(1, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$.
  4. $\mathbf{a} \leftarrow \mathfrak{pk}_a \odot \overrightarrow{\mathbf{r}} + \mathbf{e}_1$.
  5. $b \leftarrow \mathfrak{pk}_b \cdot \mathbf{r} + e_2 + (Q/P) \cdot m$.
  6. Return $(\mathbf{a}, b)$. In some applications we will require the return of $((\mathbf{a}, b), (\mathbf{r}, \mathbf{e}_1, e_2))$ in order to prove knowledge and correctness of the encryption via the ZKPoKs in Section 7.6.

TFHE.Enc$(m, \mathfrak{pk}, PKSK; P, Q, type, \hat{\ell})$:
  Here $\mathfrak{pk}$ is the public key $(\mathfrak{pk}_a, \mathfrak{pk}_b,)$ generated in the *TFHE.KeyGen* algorithm above along-side $PKSK$. The parameters $P, Q, type$ and $\hat{\ell}$ may be dropped if they can be implicitly inferred from the context. This function can be implemented entirely by the user, or by the user performing the first step and the server (which executes the homomorphic operations) performing the second step. The output is an LWE ciphertext of type *type*.
  1. $\mathfrak{ct}' \leftarrow$ *TFHE.Enc-Sub*$(m, \mathfrak{pk})$.
  2. $\mathfrak{ct} \leftarrow$ *TFHE.DimensionSwitch*$(\mathfrak{ct}', PKSK)$.
  3. Return $\mathfrak{ct}$.

TFHE.Dec$(\mathfrak{ct}, \mathfrak{sk}; P, Q)$:
  Note this works for both $ctType(\mathfrak{ct}) = LWE$ and $ctType(\mathfrak{ct}) = F\text{-}GLWE$, when $ctType(\mathfrak{ct}) = type$ (where *type* is the global variable *type* fixed during key generation), as we fixed $\mathbf{s}$ during key generation to be the appropriate key.
  1. If $ctType(\mathfrak{ct}) \neq type$ then *abort*.
  2. Write $\mathfrak{ct} = (\mathbf{a}, b)$ and $\mathfrak{sk} = (\mathbf{s}, (\overline{\mathbf{s}}_0, \ldots, \overline{\mathbf{s}}_{\overline{w}-1}))$.
  3. $\Delta \leftarrow Q/P$.
  4. $p \leftarrow b - \mathbf{a} \cdot \mathbf{s} \pmod{Q}$.
  5. p is an integer of the form $\Delta \cdot m + e$, for a "small" value of e, so we extract m by rounding to the nearest multiple of Δ.
  6. Return $m \pmod{P}$.

Figure 21: The Public Key TFHE Scheme: Encryption and Decryption.

To encrypt a message $m \in \mathbb{Z}/(P)$ (see Figure 21) we generate values $\mathbf{r}, \mathbf{e}_1$ and e_2 via the distributions $\mathbf{r} \leftarrow \{0, 1\}^{\hat{\ell}}, \mathbf{e}_1 \leftarrow TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$ and $e_2 \leftarrow TUniform(1, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$ and then we produce

$$\mathbf{a} = \mathfrak{pk}_a \odot \overrightarrow{\mathbf{r}} + \mathbf{e}_1,$$
$$b = \mathfrak{pk}_b \cdot \mathbf{r} + e_2 + \Delta \cdot m.$$

For ease of notation, when explaining our ZKPoKs later, we will let $A_{\mathfrak{pk}}$ denote the matrix representation

of the ring element $\mathfrak{pk}_a$. With this notation we have that

$$\mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} = A_{\mathfrak{pk}} \cdot \overleftrightarrow{\mathbf{r}}$$

where the multiplication on the right is the standard matrix–vector multiplication method.

The encryption method of Joye [Joy24] encrypts each message $m \in \mathbb{Z}/(P)$ as a pair $(\mathbf{a}, b)$ as above. It can be extended (see Joye [Joy24]) to encrypt up to $\hat{\ell}$ messages using a single fixed $\mathbf{a}$ value. i.e. to encrypt $m_1, \ldots, m_k$, with $k \le \ell$, one encrypts to a set $(\mathbf{a}, b_1, \ldots, b_k)$ with $b_i \in \mathbb{Z}/(Q)$. As this is not relevant to our threshold protocols we do not describe this optimization in this document.

If the mean $\mu_{\mathfrak{ct}}$ and standard deviation $\sigma_{\mathfrak{ct}}$ of the value $|p - \Delta \cdot m|$ of the input to *TFHE.Dec* satisfies

$$|\mu_{\mathfrak{ct}}| + c_{err,1} \cdot \sigma_{\mathfrak{ct}} \le \Delta/2$$

then decryption will be correct, as the following Lemma demonstrates.

Lemma 9. *The ciphertext $(\mathbf{a}, b)$ obtained by the method above can be decrypted via the usual method of computing $b - a \cdot \hat{\mathbf{s}}$.*

Proof. This follows from the equalities

$$
\begin{aligned}
b - \mathbf{a} \cdot \hat{\mathbf{s}} &= b - \left(\mathbf{a} \odot \overleftrightarrow{\hat{\mathbf{s}}} \right)_{\hat{\ell}-1}, \\
&= (\mathfrak{pk}_b \cdot \mathbf{r} + e_2 + \Delta \cdot m) - \left((\mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1) \odot \overleftrightarrow{\hat{\mathbf{s}}} \right)_{\hat{\ell}-1}, \\
&= \left(\mathfrak{pk}_b \odot \overleftrightarrow{\mathbf{r}} - (\mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1) \odot \overleftrightarrow{\hat{\mathbf{s}}} \right)_{\hat{\ell}-1} + e_2 + \Delta \cdot m, \\
&= \left((\mathfrak{pk}_a \odot \overleftrightarrow{\hat{\mathbf{s}}} + \mathbf{e}) \odot \overleftrightarrow{\mathbf{r}} - \mathfrak{pk}_a \odot \overleftrightarrow{\mathbf{r}} \odot \overleftrightarrow{\hat{\mathbf{s}}} - \mathbf{e}_1 \odot \overleftrightarrow{\hat{\mathbf{s}}} \right)_{\hat{\ell}-1} + e_2 + \Delta \cdot m, \\
&= \left(\mathbf{e} \odot \overleftrightarrow{\mathbf{r}} - \mathbf{e}_1 \odot \overleftrightarrow{\hat{\mathbf{s}}} \right)_{\hat{\ell}-1} + e_2 + \Delta \cdot m, \\
&= e' + \Delta \cdot m.
\end{aligned}
$$

$\square$

For later, we will need to estimate the mean and standard deviation of the value $|p - \Delta \cdot m|$ of the output of the *TFHE.Enc* procedure. As noted above the *TFHE.DimensionSwitch* operation does not affect the mean, but it does add the term σ_{DS}^2 onto the variance.

When analysing *TFHE.Enc-Sub* we need to consider that the adversary may be the encryptor and thus may try to encrypt using too much noise. If used with the ZKPoKs from Section 7.6, the ZKPoK prover is applied by the encryptor to the output of *TFHE.Enc-Sub* to obtain the proof $\pi_{\mathfrak{ct}'}$ on the ciphertext $\mathfrak{ct}'$. The pair $(\mathfrak{ct}', \pi_{\mathfrak{ct}'})$ are passed to the entity who will use the ciphertext. This entity can then verify the proof, and then apply *TFHE.DimensionSwitch* to the ciphertext $\mathfrak{ct}'$ in order to obtain the output of *TFHE.Enc*. Thus the worst that the adversary can do, to maximise the noise in a ciphertext, is to pick encryption noise values of the form $\mathbf{r} = \mathbf{1}$, and noise values $(\mathbf{e}_1|e_2) \le zk\text{-}slack \cdot 2^{b_{\hat{\ell}}}$, where $zk\text{-}slack$ is the soundness slack value introduced in Section 4.11. This enables us to derive worst case expected values for the mean and standard deviation of

$$e' = b - \mathbf{a} \cdot \hat{\mathbf{s}} - \Delta \cdot m = (\mathbf{e} \odot \overleftrightarrow{\mathbf{r}} - \mathbf{e}_1 \odot \overleftrightarrow{\hat{\mathbf{s}}})_{\hat{\ell}-1} + e_2$$

assuming the public key has been honestly generated, i.e. $Var(s_i) = 1/4$ and $Exp(s_i) = 1/2$,

$$\mu_{PKE} = Exp[e'] \quad = \quad \sum_{i=1}^{\hat{\ell}} Exp(\mathbf{e}) + zk\text{-}slack \cdot \sum_{i=1}^{\hat{\ell}} 2^{b_{\hat{\ell}}} \cdot Exp(s_i) + zk\text{-}slack \cdot 2^{b_{\hat{\ell}}}$$

$$\begin{aligned} &= \quad 0 + \textit{zk-slack} \cdot \hat{\ell} \cdot 2^{b_{\hat{\ell}}} \cdot \frac{1}{2} + \textit{zk-slack} \cdot 2^{b_{\hat{\ell}}} \\ &= \quad \textit{zk-slack} \cdot \left(\frac{\hat{\ell}}{2} + 1 \right) \cdot 2^{b_{\hat{\ell}}}, \end{aligned} \tag{12}$$

$$\begin{aligned} \sigma^2_{PKE} = Var[e'] \quad &= \quad \sum_{i=1}^{\hat{\ell}} Var(\mathbf{e}) + \textit{zk-slack}^2 \cdot \sum_{i=1}^{\hat{\ell}} 2^{2 \cdot b_{\hat{\ell}}} \cdot Var(s_i) \\ &= \quad \hat{\ell} \cdot \sigma^2_{b_{\hat{\ell}}} + \textit{zk-slack}^2 \cdot \hat{\ell} \cdot 2^{2 \cdot b_{\hat{\ell}}} \cdot \frac{1}{4} \\ &= \quad \hat{\ell} \cdot \left(\sigma^2_{b_{\hat{\ell}}} + \textit{zk-slack}^2 \cdot 2^{2 \cdot b_{\hat{\ell}} - 2} \right), \end{aligned} \tag{13}$$

where $\sigma_{b_{\hat{\ell}}} = \sqrt{(2^{2 \cdot b_{\hat{\ell}} + 1} + 1)/6}$ is the standard deviation of the distribution $TUniform(1, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$.

5.6.3.6 Homomorphic Operations: The basic homomorphic operations in TFHE are addition and scalar multiplication as described in Figure 22. They consist of simply adding ciphertexts component–wise and scaling each ciphertext component by the scalar, respectively. All operations are performed modulo Q on ciphertexts of type *type*. The addition of two ciphertexts results in an addition of the mean and variance of the quantity $|p - \Delta \cdot m|$.

The Basic Homomorphic Operations: Addition and Scalar Multiplication

TFHE.Add($\mathfrak{ct}_1, \mathfrak{ct}_2$):
    We have input LWE/F–GLWE ciphertexts $\mathfrak{ct}_1$ and $\mathfrak{ct}_2$ encrypting messages $m_1 \in \mathbb{Z}/(P)$ and $m_2 \in \mathbb{Z}/(P)$, respectively. The output LWE/F–GLWE ciphertext is an encryption of $m_1 + m_2$.
    1. Write $\mathfrak{ct}_1 = (\mathbf{a}_1, b_1)$.
    2. Write $\mathfrak{ct}_2 = (\mathbf{a}_2, b_2)$.
    3. $\mathbf{a}' \leftarrow \mathbf{a}_1 + \mathbf{a}_2$.
    4. $b' \leftarrow b_1 + b_2$.
    5. Return $(\mathbf{a}', b')$.

TFHE.ScalarMult($\alpha, \mathfrak{ct}$):
    We have $\mathfrak{ct}$ an LWE/F–GLWE ciphertext encrypting $m \in \mathbb{Z}/(P)$ and $\alpha \in \mathbb{Z}/(P)$; where recall $\mathbb{Z}/(P) = \{-P/2 + 1, \dots, P/2\}$. The output LWE/F–GLWE ciphertext is an encryption of $\alpha \cdot m$.
    1. Write $\mathfrak{ct} = (\mathbf{a}, b)$.
    2. Consider α as an integer (i.e. in $\mathbb{Z}$).
    3. $\mathbf{a}' \leftarrow \alpha \cdot \mathbf{a}$ (i.e. scalar multiplication of the vector $\mathbf{a}$ by the scalar α.
    4. $b' \leftarrow \alpha \cdot b$.
    5. Return $(\mathbf{a}', b')$.

Figure 22: The Basic Homomorphic Operations: Addition and Scalar Multiplication.

Further homomorphic operations in TFHE are enabled via the Programmable Bootstrapping (PBS) operation. This function, which we shall describe in this subsection, enables two operations to be performed at once: firstly the noise term in the FHE ciphertext is reduced, and secondly a negacyclic function is applied to the message.

Recall for TFHE the plaintext space is $\mathbb{Z}/(P)$ where P is a power of two. A negacyclic function $f : \mathbb{Z}/(P) \longrightarrow \mathbb{Z}/(P)$ is one which satisfies

$$f(x + P/2) = -f(x)$$

for all $x \in \mathbb{Z}/(P)$. During homomorphic operations we may want to evaluate a function $g : S \longrightarrow \mathbb{Z}/(P)$ on a subset $S \subseteq \mathbb{Z}/(P)$. We can evaluate g during the bootstrap operation "for free", if there is a negacyclic function f whose domain is the whole of $\mathbb{Z}/(P)$ which is identical to g on the domain of g, i.e. $g(x) = f(x)$

for all $x \in S$. Typically, we will want to apply an arbitrary function $g : [0, \ldots, P/2 - 1] \longrightarrow \mathbb{Z}/(P)$ so we simply define $f : \mathbb{Z}/(P) \longrightarrow \mathbb{Z}/(P)$ as

$$
f(x) = \begin{cases} g(x) & \text{if } 0 \leq x < P/2, \\ -g(x - P/2) & \text{if } P/2 \leq x < P, \end{cases}
$$

and perform the PBS using f. Although sometimes efficiencies can be obtained by considering even larger domains for g, if a suitable negacyclic function f can be defined which matches g on the required domain of g. We say f is a *negacyclic extension of* the function g.

We let $PBS_f(\mathfrak{ct})$ denote the application of a PBS operation on a ciphertext $\mathfrak{ct}$ with respect to a negacyclic function f. To see how this enables basic homomorphic operations, over bits, consider a plaintext space with $P = 2^3$. We let $xyz \in \mathbb{Z}/(P)$, for bits x, y and z, denote the value $x \cdot 2^2 + y \cdot 2 + z$. If we assume that single bits are encrypted in input ciphertexts $\mathfrak{ct}_1$ and $\mathfrak{ct}_2$ then homomorphic addition modulo two, i.e. the XOR operation, can be computed by performing

$$PBS_f(\mathfrak{ct}_1 + \mathfrak{ct}_2)$$

for a suitable negacyclic extension f of the function $g : \{0, 1, 2\} \longrightarrow \mathbb{Z}/(P)$ given by $g(x) = x \pmod 2$. Thus even though g is not negacyclic on the whole of $\mathbb{Z}/(P)$ we can still use the PBS operation to compute it homomorphically on the domain required.

The multiplication operation modulo two, i.e. the AND operation, can be computed by performing

$$PBS_f(\mathfrak{ct}_1 + \mathfrak{ct}_2)$$

where f is the negacyclic extension of the function g which maps the input bit string xyz to $00y$. Again, we only need to apply g on the input domain $\{0, 1, 2\}$ so PBS_f correctly computes the AND operation.

Clearly, by having different plaintext space sizes, and utilizing different functions f in the PBS operation one can obtain additional performance enhancements. We do not consider any of these optimizations here, as they are best left to the application layer.

The PBS is the main operation in TFHE, it can be divided into three main steps: a modulus switch, a chain of external products and a flatten operation. Most of the time, the PBS is used in combination with a key switch which can be either placed after the bootstrap (as in original TFHE paper when we have $type = LWE$) or before (as proposed in [BMMP18] for when we have $type = F\text{-}GLWE$).

5.6.3.7 Key Switch: The KeySwitch operation takes a ciphertext of type *F-GLWE* and outputs cipher–text of type *LWE* encrypting the same message. One can see that it is highly similar to the operation *TFHE.DimensionSwitch* given earlier, a fact which we will return to below. The algorithm is presented in Figure 23. The associated variance of the output $|p - \Delta \cdot m|$ is modified by the addition of the term (again see [CLOT21] for a full derivation)

$$
\begin{aligned}
\sigma_{KS}^2 &= w \cdot N \cdot \left(\frac{Q^2}{12 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} - \frac{1}{12} \right) \cdot (Var(s_i) + Exp^2(s_i)) \\
&\quad + \frac{w \cdot N}{4} \cdot Var(s_i) + w \cdot N \cdot \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12} \right) \\
&= \frac{w \cdot N}{2} \cdot \left(\frac{Q^2}{12 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} - \frac{1}{12} \right) \\
&\quad + w \cdot N \cdot \left(\frac{1}{16} + \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12} \right) \right)
\end{aligned}
$$

$$= w \cdot N \cdot \left(\frac{Q^2}{24 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} + \frac{1}{48} + \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12} \right) \right) \tag{14}$$

since for a binary secret key we have $Var(s_i) = 1/4$ and $Exp(s_i) = 1/2$, where

$$\sigma_{ksk} = \sigma_{b_\ell} = \sqrt{(2^{2 \cdot b_\ell + 1} + 1)/6}.$$

TFHE Key Switching

$TFHE.KeySwitch(\mathfrak{ct}, KSK; Q, \ell, N, w, \beta_{ksk}, \nu_{ksk})$:
    The input ciphertext is an F–GLWE ciphertext of dimension $w \cdot N$, the output ciphertext is an LWE ciphertext of dimension ℓ.
    1. If $ctType(\mathfrak{ct}) \neq F\text{-}GLWE$ then *abort*.
      This is purely a sanity check, and can be ignored in correct implementations.
    2. Write $\mathfrak{ct} = (\mathbf{a}, b)$.
    3. Parse KSK as $\{KSK_{i,j}\}_{i,j}$ for $i \in [0, \ldots, w-1]$ and $j \in [0, \ldots, N-1]$.
    4. For $i \in [0, \ldots, w \cdot N - 1]$ do:
      (a) $\mathbf{d}_i \leftarrow TFHE.Decompose(\mathbf{a}[i], Q, \beta_{ksk}, \nu_{ksk})$.
    5. For $i \in [0, \ldots, w-1]$ do:
      (a) For $j \in [0, \ldots, N-1]$ do:
        i. Parse $KSK_{i,j}$ as $(KSK_{i,j,0}, \ldots, KSK_{i,j,\nu_{ksk}-1})$ and parse $KSK_{i,j,k}$ as $(KSK_{i,j,k}^a, KSK_{i,j,k}^b)$ for each $k \in [0, \ldots, \nu_{ksk} - 1]$.
        ii. Define the ν_{ksk}–by–ℓ matrix

$$M_{i,j} \leftarrow \begin{pmatrix} KSK_{i,j,0}^a[0] & \ldots & KSK_{i,j,0}^a[\ell-1] \\ \vdots & & \vdots \\ KSK_{i,j,\nu_{ksk}-1}^a[0] & \ldots & KSK_{i,j,\nu_{ksk}-1}^a[\ell-1] \end{pmatrix},$$

        where $KSK_{i,j,k}^a[t]$ is the t–th element of the vector $KSK_{i,j,k}^a$.
        iii. $\mathbf{v}_{i,j} \leftarrow (KSK_{i,j,0}^b, \ldots, KSK_{i,j,\nu_{ksk}-1}^b)$, i.e. a ν_{ksk}–dimensional vector.
    6. $\mathbf{a}' \leftarrow -\sum_{i=0}^{w-1} \sum_{j=0}^{N-1} \mathbf{d}_{i \cdot N + j} \cdot M_{i,j}$.
    Note this is a sum of $w \cdot N$ vector by matrix products.
    7. $b' \leftarrow b - \sum_{i=0}^{w-1} \sum_{j=0}^{N-1} \mathbf{d}_{i \cdot N + j} \cdot \mathbf{v}_{i,j}$.
    Note this is a sum of $w \cdot N$ vector by vector inner products.
    8. Return $(\mathbf{a}', b')$.

Figure 23: TFHE Key Switching Algorithm mapping a F–GLWE ciphertext to an LWE one.

We now return to the similarity between $TFHE.KeySwitch$ and $TFHE.DimensionSwitch$. First note, that one can view the output of $TFHE.Enc$ as a flattened GLWE ciphertext of dimension $\hat{\ell}$. In particular, if we let $\zeta_M : \mathbf{R}^{(M)} \rightarrow \mathbf{R}^{(M)}$ be the automorphism which implements the map $X \mapsto X^{-1}$, then $\mathbf{a}'$ in Figure 13 is $(\zeta_N(\mathbf{a}_0), \ldots, \zeta_N(\mathbf{a}_{w-1}))$. Using the notation from Figure 21, we can define $\hat{\mathbf{e}} \leftarrow \zeta_{\hat{\ell}}(\mathbf{e})$, $\hat{\mathbf{e}}_1 \leftarrow \zeta_{\hat{\ell}}(\mathbf{e}_1)$, $\hat{\mathfrak{pk}}_b \leftarrow \zeta_{\hat{\ell}}(\mathfrak{pk}_b), \hat{\mathfrak{pk}}_a \leftarrow -X \cdot \zeta_{\hat{\ell}}(\mathfrak{pk}_a)$ and let $\hat{\mathbf{e}}_2$ be such that $\hat{\mathbf{e}}_2[0] = e_2$. Then we have that $\hat{\mathfrak{pk}}_b = \hat{\mathfrak{pk}}_a \odot \hat{\mathbf{s}} + \hat{\mathbf{e}}$. Further, defining $\hat{\mathbf{a}} \leftarrow \hat{\mathfrak{pk}}_a \odot \mathbf{r} + \hat{\mathbf{e}}_1$ and $\hat{\mathbf{b}} \leftarrow \hat{\mathfrak{pk}}_b \odot \mathbf{r} + \hat{\mathbf{e}}_2 + (Q/P) \cdot m$ we have the equality

$$(\mathbf{a}, b, \sigma_{PKE}) = TFHE.Flatten((\hat{\mathbf{a}}, \hat{\mathbf{b}}, \sigma_{PKE}), \hat{\ell}, 1).$$

Here, $(\hat{\mathbf{a}}, \hat{\mathbf{b}}, \sigma_{PKE})$ is an RLWE ciphertext encrypting the constant polynomial m under the secret key $\hat{\mathbf{s}}$, i.e. $\hat{\mathbf{b}} = \hat{\mathbf{a}} \odot \hat{\mathbf{s}} + \mathbf{e}' + (Q/P) \cdot m$ for some $\mathbf{e}' = \hat{\mathbf{e}} \odot \mathbf{r} - \hat{\mathbf{e}}_1 \odot \hat{\mathbf{s}} + \hat{\mathbf{e}}_2$. Thus we can apply $TFHE.KeySwitch$ to a fresh encryption in order to change its LWE dimension from $\hat{\ell}$ to d (where d is either ℓ or $w \cdot N$) by using the key $PKSK$. In particular, given the output $\mathfrak{ct}$ from $TFHE.Enc$ the following two calls are (essentially,

up to the parsing of the data structure *PKSK*) identical

$$\mathfrak{ct}' \leftarrow TFHE.KeySwitch(\mathfrak{ct}, PKSK; Q, d, 1, \hat{\ell}, \beta_{pksk}, \nu_{pksk}),$$

$$\mathfrak{ct}' \leftarrow TFHE.DimensionSwitch(\mathfrak{ct}, PKSK; Q, d, \hat{\ell}, \beta_{pksk}, \nu_{pksk}).$$

5.6.3.8 Modulus Switch: This operation, presented in Figure 24, takes an LWE ciphertext, with ciphertext modulus Q, and switches it to an LWE ciphertext with modulus $2 \cdot N$. This algorithm is the first stage of bootstrapping. To ensure the bootstrapping operation is correct, we need to ensure the output of the modulus switch will not produce a decryption error. The result will be correct (with probability at least $1 - err$) if we have that

$$|\mu_{\mathfrak{ct}}| + c_{err,1} \cdot \sqrt{\sigma_{\mathfrak{ct}}^2 + \sigma_{MS}^2} < \frac{\Delta}{2} \tag{15}$$

where $\mu_{\mathfrak{ct}}$ and $\sigma_{\mathfrak{ct}}$ are estimate of the mean and standard deviation of the noise term $|p - \Delta \cdot m|$ in the input ciphertext and (again see [CLOT21])

$$\sigma_{MS}^2 = \frac{Q^2}{48 \cdot N^2} - \frac{1}{12} + \ell \cdot \left(\frac{Q^2}{96 \cdot N^2} + \frac{1}{48} \right). \tag{16}$$

One can reduce this error probability to any desired value by increasing the value of $c_{err,1}$. Note, we will sometimes call *TFHE.ModSwitch*, see *TFHE.SwitchSquash* below, with the larger parameter of $\overline{N}$, in which case one should replace N with $\overline{N}$ in equations (15) and (16).

TFHE Modulus Switching

TFHE.ModSwitch$(\mathfrak{ct}; Q, \ell, N)$:
  The input LWE ciphertext (of dimension ℓ) is with respect to ciphertext modulus Q, the output LWE ciphertext is with respect to ciphertext modulus $2 \cdot N$.
  1. If $ctType(\mathfrak{ct}) \neq LWE$ then *abort*.
     This is purely a sanity check, and can be ignored in correct implementations.
  2. Write $\mathfrak{ct} = (\mathbf{a}, b)$.
  3. Think of $\mathbf{a}$ (resp. b) as in $\mathbb{Z}^{\ell}$ (resp. $\mathbb{Z}$) and not $(\mathbb{Z}/(Q))^{\ell}$ (resp. $\mathbb{Z}/(Q)$).
  4. $\mathbf{a}' \leftarrow \lceil \mathbf{a} \cdot 2 \cdot N/Q \rfloor$.
  5. $b' \leftarrow \lceil b \cdot 2 \cdot N/Q \rfloor$.
  6. Return $(\mathbf{a}', b')$.

Figure 24: TFHE Modulus Switching Algorithm from Q to $2 \cdot N$.

This is the place for the TFHE algorithm where the error probability has the most effect on the underlying parameters. By modifying the modulus switching algorithm slightly, see for example the method in [HKLS24] or [BJSW24], one can obtain significantly better parameters (and hence significantly better overall performance) whilst still maintaining the same error probability of $err = 2^{-128}$. Such optimizations are beyond the scope of this document, which purely aims to present the basic FHE algorithms in order to describe the resulting threshold protocols fully.

5.6.3.9 Bootstrap: Bootstrapping takes a ciphertext of type *LWE* and outputs a ciphertext of type *F-GLWE* but with (potentially) smaller noise, see Figure 26 for the pseudo–code. It is here we apply the negacyclic function f. This function calls a sub–procedure for evaluating an external product (see Figure 25).

The first thing a bootstrap operations performs is a modulus switch, therefore the input to the bootstrap operation (for it to be correct with a given probability) must satisfy equation (15). The noise

---

TFHE External Product

TFHE.ExternalProduct$(\mathfrak{ct}, CT; Q, \beta, \nu)$:

This takes as input a *GLWE* ciphertexts $\mathfrak{ct}$ encrypting **m** and a *GGSW* ciphertext CT (with respect to β and ν) which encrypts a bit m, both under the secret key $(\mathbf{s}_0, \ldots, \mathbf{s}_{w-1})$. It outputs a *GLWE* encryption of $m \cdot \mathbf{m}$ under $(\mathbf{s}_0, \ldots, \mathbf{s}_{w-1})$.

1. Parse $\mathfrak{ct}$ as $(\boldsymbol{\alpha}_0, \ldots, \boldsymbol{\alpha}_{w-1}, \mathbf{b})$.
2. For $j \in [0, \ldots, N-1]$ do:
    (a) For $i \in [0, \ldots, w-1]$ do:
        i. $\mathbf{d}_{i,j} \leftarrow$ *TFHE.Decompose*$(\boldsymbol{\alpha}_i[j], Q, \beta, \nu)$.
    (b) $\mathbf{d}_{w,j} \leftarrow$ *TFHE.Decompose*$(\mathbf{b}[j], Q, \beta, \nu)$.
3. For $i \in [0, \ldots, w]$ do:
    (a) $\mathbf{D}_i \leftarrow \left(\sum_{j=0}^{N-1} \mathbf{d}_{i,j}[0] \cdot X^j, \ldots, \sum_{j=0}^{N-1} \mathbf{d}_{i,j}[\nu-1] \cdot X^j \right)$, where X is the ring indeterminate of $\mathbf{R}$.
4. Parse CT as $(\mathfrak{ct}'_0, \ldots, \mathfrak{ct}'_w)$.
5. For $i \in [0, \ldots, w]$ do:
    (a) Parse $\mathfrak{ct}'_i$ as $(\mathfrak{ct}'_{i,0}, \ldots, \mathfrak{ct}'_{i,\nu-1})$ and parse $\mathfrak{ct}'_{i,j}$ as $(\boldsymbol{\alpha}'_{i,j,0}, \ldots, \boldsymbol{\alpha}'_{i,j,w-1}, \mathbf{b}'_{i,j})$ for $j \in [0, \ldots, \nu-1]$.
    (b) Define the ν-by-$(w+1)$ matrix over $\mathbf{R}_Q$

$$CT_i \leftarrow \begin{pmatrix} \boldsymbol{\alpha}'_{i,0,0} & \cdots & \boldsymbol{\alpha}'_{i,0,w-1} & \mathbf{b}'_{i,0} \\ \vdots & & \vdots & \vdots \\ \boldsymbol{\alpha}'_{i,\nu-1,0} & \cdots & \boldsymbol{\alpha}'_{i,\nu-1,w-1} & \mathbf{b}'_{i,\nu-1} \end{pmatrix}.$$

6. Return $\sum_{i=0}^{w} \mathbf{D}_i \cdot CT_i$ where the vector–matrix multiplication is performed over $\mathbf{R}_Q$.

Figure 25: TFHE External Product Evaluation.

output from bootstrap has mean zero and variance σ_{BS}^2 where (again see [CLOT21])

$$\sigma_{BS}^2 = \ell \cdot \left(\quad \nu_{bk} \cdot (w+1) \cdot N \cdot \left(\frac{\beta_{bk}^2 + 2}{12} \right) \cdot \sigma_{bk}^2 \right.$$
$$+ \left(\frac{Q^2 - \beta_{bk}^{2 \cdot \nu_{bk}}}{24 \cdot \beta_{bk}^{2 \cdot \nu_{bk}}} \right) \cdot \left(1 + \frac{w \cdot N}{2} \right) \tag{17}$$
$$\left. + \frac{w \cdot N}{32} + \frac{1}{16} \cdot \left(1 - \frac{w \cdot N}{2} \right)^2 \quad \right)$$

with $\sigma_{bk} = \sqrt{(2^{2 \cdot b_{w \cdot N} + 1} + 1)/6}$. An important requirement for protocols is that the bootstrapping operation is deterministic.

TFHE Bootstrapping

TFHE.BootStrap$(\mathfrak{ct}, f, BK; P, Q, Q', \ell, N, w, \beta_{bk}, \nu_{bk})$:

This takes an input LWE ciphertext $\mathfrak{ct} \in (\mathbb{Z}/(Q))^{\ell+1}$ encrypting a message m, and produces an F–GLWE ciphertext in $(\mathbb{Z}/(Q'))^{w \cdot N+1}$ encrypting the message $f(m)$. It uses the boot–strapping key BK defined with respect to the modulus Q'. It assumes f is negacyclic on the plaintext domain of $\mathfrak{ct}$.

1. If $ctType(\mathfrak{ct}) \neq LWE$ then *abort*.
   This is purely a sanity check, and can be ignored in correct implementations.
2. Parse BK as $\{BK_i\}_i$ for $i \in [0, \dots, \ell - 1]$.
3. Define $\mathbf{R}$ as the ring of degree N with ring indeterminate X.
4. $\Delta \leftarrow Q'/P$.
   Note this might not be the global value of Δ.
5. $(\tilde{\mathbf{a}}, \tilde{b}) \leftarrow$ *TFHE.ModSwitch*$(\mathfrak{ct}; Q, \ell, N)$.
6. $B \leftarrow X^{-\tilde{b}} \cdot \sum_{j=0}^{N-1} \Delta \cdot f(\lceil j \cdot P/(2 \cdot N) \rceil) \cdot X^j$.
7. Define the trivial GLWE ciphertext $\mathfrak{ct}' \leftarrow (0, 0, \dots, 0, B) \in \mathbf{R}_{Q'}^{w+1}$.
8. For $i \in [0, \dots, \ell - 1]$ do:
   (a) $\hat{\mathfrak{ct}} \leftarrow ((X^{\tilde{\mathbf{a}}[i]} - 1) \cdot \mathfrak{ct}'[0], \dots, (X^{\tilde{\mathbf{a}}[i]} - 1) \cdot \mathfrak{ct}'[w])$.
   (b) $\mathfrak{ct}' \leftarrow$ *TFHE.Add*$\left($*TFHE.ExternalProduct*$(\hat{\mathfrak{ct}}, BK_i, Q', \beta_{bk}, \nu_{bk}), \mathfrak{ct}'\right)$.
9. Return (*TFHE.Flatten*$(\mathfrak{ct}', N, w), \sigma_{BS})$, where σ_{BS} is computed from equation (17) using the values of $(\ell, Q', N, w, \beta_{bk}, \nu_{bk})$ used in this call.

Figure 26: TFHE Bootstrapping Algorithm Applying the Negacyclic Function f.

5.6.3.10 PBS: The full PBS operation is the combination of bootstrap and key switch, see Figure 27. We shall refer to this operation by the notation $(\mathbf{a}', b') \leftarrow PBS((\mathbf{a}, b), f, PK)$, or sometimes more succinctly by $\mathfrak{ct}' \leftarrow PBS_f(\mathfrak{ct})$. As such the operation will be correct (with a given probability) only if the input ciphertext noise, at the point of the internal *TFHE.BootStrap* call, satisfies equation (15). The order of application of bootstrap and key switch depends on whether the input ciphertext is of type *LWE* or *F-GLWE*.

TFHE Full PBS

TFHE.PBS$(\mathfrak{ct}, f, PK; P, Q, \ell, N, w, \beta_{ksk}, \beta_{bk}, \nu_{ksk}, \nu_{bk})$:

This takes an input LWE/F–GLWE ciphertext encrypting a message m, and outputs an LWE/F–GLWE ciphertext encrypting the message $f(m)$. It assumes f is negacyclic on the plaintext domain of $\mathfrak{ct}$, and that $ctType(\mathfrak{ct}) = type$.

1. Parse PK as $(\mathfrak{pk}, PKSK, KSK, BK)$.
2. If $ctType(\mathfrak{ct}) = LWE$ then
   (a) $\hat{\mathfrak{ct}} \leftarrow$ *TFHE.BootStrap*$(\mathfrak{ct}, f, BK; P, Q, Q, \ell, N, w, \beta_{bk}, \nu_{bk})$.
   (b) $\mathfrak{ct}' \leftarrow$ *TFHE.KeySwitch*$(\hat{\mathfrak{ct}}, KSK; Q, \ell, N, w, \beta_{ksk}, \nu_{ksk})$.
3. Else
   (a) $\hat{\mathfrak{ct}} \leftarrow$ *TFHE.KeySwitch*$(\mathfrak{ct}, KSK; Q, \ell, N, w, \beta_{ksk}, \nu_{ksk})$.
   (b) $\mathfrak{ct}' \leftarrow$ *TFHE.BootStrap*$(\hat{\mathfrak{ct}}, f, BK; P, Q, Q, \ell, N, w, \beta_{bk}, \nu_{bk})$.
4. Return $\mathfrak{ct}'$.

Figure 27: The Full TFHE PBS Algorithm.

5.6.3.11 Switch–n–Squash: Switch–n–Squash is a special kind of bootstrap, see Figure 28. The underlying bootstrapping key is assumed to be for an input ciphertext of type *LWE*, so we need to do a key switch if the input ciphertext is of type *F-GLWE*. We shall refer to this operation by the notation $(\mathbf{a}, b) \leftarrow$ *TFHE.SwitchSquash*$((\mathbf{a}', b'), \overline{BK})$. As such the operation will be correct (with a given probability) only

if the input noise satisfies equation (15) with σ_{MS} a function of $\overline{N}, Q$ and ℓ, i.e. we use

$$\overline{\sigma_{MS}}^2 = \frac{Q^2}{48 \cdot \overline{N}^2} - \frac{1}{12} + \ell \cdot \left(\frac{Q^2}{96 \cdot \overline{N}^2} + \frac{1}{48} \right). \tag{18}$$

If the noise does not satisfy this requirement then the call to *TFHE.ModSwitch* will abort. In this algorithm, we need an additional bootstrapping key $\overline{BK}$, which was computed as part of the key generation operation when the flag $\overline{flag}$ is set at the point of calling *TFHE.KeyGen*.

TFHE Switch–n–Squash

TFHE.SwitchSquash($\mathfrak{ct}, \overline{BK}; P, Q, \overline{Q}, \ell, \overline{N}, \overline{w}, \overline{\beta_{bk}}, \overline{v_{bk}}$) :
   This takes an input LWE/F–GLWE ciphertext encrypting a message m and a ciphertext modulus Q. It then outputs a F–GLWE ciphertext encrypting the message m with an LWE dimension of $\overline{\ell} = \overline{w} \cdot \overline{N}$ and a ciphertext modulus $\overline{Q}$.
   1. If $ctType(\mathfrak{ct}) = F\text{-}GLWE$ then $\mathfrak{ct} \leftarrow TFHE.KeySwitch(\mathfrak{ct}, KSK; Q, \ell, N, w, \beta_{ksk}, v_{ksk})$.
   2. $\mathfrak{ct}' \leftarrow TFHE.BootStrap(\mathfrak{ct}, f, \overline{BK}; P, Q, \overline{Q}, \ell, \overline{N}, \overline{w}, \overline{\beta_{bk}}, \overline{v_{bk}})$, where f is the identity function and is negacyclic on the plaintext domain of $\mathfrak{ct}$.
   3. Return $\mathfrak{ct}'$.

Figure 28: The TFHE Switch–n–Squash Algorithm.

The *TFHE.SwitchSquash* operation produces an LWE ciphertext $\overline{\mathfrak{ct}}$, encrypting the same message $m \in \mathbb{Z}/(P)$, however the ciphertext has an $\boldsymbol{\alpha}$ component of dimension $\overline{\ell} = \overline{w} \cdot \overline{N}$ (being the output of flattening a GLWE ciphertext of dimension $\overline{w}$ over the ring of dimension $\overline{N}$), and uses a ciphertext modulus $\overline{Q}$. The standard deviation of the noise term $|p' - \Delta \cdot m|$ in the output ciphertext is $\overline{\sigma_{BS}}$, where $\overline{\sigma_{BS}}$ is given by equation (17), but with each variable $\cdot$ replaced by it's $\overline{\cdot}$ version (except for the variable ℓ), i.e.

$$\begin{aligned}
\overline{\sigma_{BS}}^2 = \ell \cdot \Bigg(\quad & \overline{v_{bk}} \cdot (\overline{w} + 1) \cdot \overline{N} \cdot \left(\frac{\overline{\beta_{bk}}^2 + 2}{12} \right) \cdot \overline{\sigma_{bk}}^2 \\
& + \left(\frac{\overline{Q}^2 - \overline{\beta_{bk}}^{2 \cdot \overline{v_{bk}}}}{24 \cdot \overline{\beta_{bk}}^{2 \cdot \overline{v_{bk}}}} \right) \cdot \left(1 + \frac{\overline{w} \cdot \overline{N}}{2} \right) \\
& + \frac{\overline{w} \cdot \overline{N}}{32} + \frac{1}{16} \cdot \left(1 - \frac{\overline{w} \cdot \overline{N}}{2} \right)^2 \quad \Bigg)
\end{aligned}$$

where $\overline{\sigma_{bk}} = \sqrt{(2^{2 \cdot b_{\overline{w} \cdot \overline{N}} + 1} + 1)/6}$. The output mean of $|p' - \Delta \cdot m|$ is zero.

5.6.3.12 Optimization Using FFT: The main computational costs in our TFHE operations is line 6 of Figure 25 which computes a sum of products of terms in $\mathbf{R}_Q$. Recall this computes a ring multiplication, which if we process it efficiently, via the FFT based multiplication in Figure 4, will result in an error vector being introduced. Luckily, as we are using a Type–II ciphertext representation for TFHE, the error introduced by the FFT and inverse–FFT operations gets "absorbed" into the noise term in the FHE ciphertext. We simply need to find a way of measuring the extra noise resulting from these operations, and then account for it within our noise formulae. In [Tap23][Chapter 3] experiments are carried out in order to determine an approximate formula for the resulting additional noise term in the entire bootstrapping operation. This is determined to be

$$FFTNoise(Q, \ell, w, N, \beta_{bk}, v_{bk}) = 2^v \cdot \ell \cdot (w + 1) \cdot N^2 \cdot \beta_{bk}^2 \cdot v_{bk}.$$

The value v (which is a function of Q) depends on the precision being used. With a mantissa of b bits we have $v = 2 \cdot (\log_2 Q - b) - 2.6$; so for $Q = 2^{64}$ and using $float64$ we find $v = 19.4$, and for $Q = 2^{128}$ and using $float128$ we find $v = 41.4$.

If the FFT algorithm is used to implement 6 of Figure 25, then $FFTNoise(Q, \ell, w, N, \beta_{bk}, \nu_{bk})$ (resp. $FFTNoise(\overline{Q}, \ell, \overline{w}, \overline{N}, \overline{\beta_{bk}}, \overline{\nu_{bk}})$) needs to be added onto the value of σ_{BS} (resp. $\overline{\sigma_{BS}}$) when computing the standard deviation of noise when exiting algorithm $TFHE.BootStrap$.

5.6.3.13 Admissible Linear Functions: In analyzing correctness for BGV we assumed a layered circuit of multiplication operations followed by λ fan–in addition gates. We utilize a similar analysis here, however we instead allow a restricted class of linear operations, followed by a PBS operation. We define a set of *admissible linear functions* $\mathcal{A}$. Each function $h \in \mathcal{A}$ is defined by a vector $\boldsymbol{\alpha} \in \mathbb{Z}/(P)^t$ for some (function specific) value of t. The function h on ciphertexts represented by the vector $\boldsymbol{\alpha}$ is a function of t ciphertexts $\mathbf{c} = (\mathfrak{ct}_i)_{i=1}^{t}$. We have $h(\mathbf{c})$ being given by the dot–product $\boldsymbol{\alpha} \cdot \mathbf{c}$ (i.e. by executing t $TFHE.ScalarMult$ operations and $t-1$ $TFHE.Add$ operations)

For a given set of admissible functions $\mathcal{A}$ in an application we define the application constant

$$\lambda = \max_{\boldsymbol{\alpha} \in \mathcal{A}} \|\boldsymbol{\alpha}\|_2.$$

The reason for taking the two–norm here is that we use λ to bound the expansion in the noise term after applying a function $h \in \mathcal{A}$. This is because if we take two independent centered Gaussian distributions X_0, X_1 of standard deviation σ and two integers ω_0, ω_1, the variance of the random variable $Y = \omega_0 \cdot X_0 + \omega_1 \cdot X_1$ is $(\omega_0^2 + \omega_1^2) \cdot \sigma^2 = \|(\omega_0, \omega_1)\|_2^2 \cdot \sigma^2$.

We now present some examples here of plaintext spaces of interest and the associated sets of admissible linear functions, as well as PBS operations.

$P = 2^3$ and Binary Operations: To execute binary circuits one can select a plaintext space of 3–bits, with binary inputs and outputs being encoded as the bit sequences $00x$. We can add two, three, or four such values using the linear functions $(1, 1)$, $(1, 1, 1)$ and $(1, 1, 1, 1)$. The result modulo two, i.e. the XOR operation, can then be obtained by applying the PBS operation with the negacyclic function $f_\oplus$ which maps $x \longrightarrow x \pmod 2$ when $x \in \{0, 1, 2, 3\}$, and $x \longrightarrow -(x \pmod 2)$ when $x \in \{4, 5, 6, 7\}$. Note that, this gives the correct reduction when we perform four additions followed by the PBS, since $-0 = 0$. However, if we performed five additions before applying the PBS then the output would not necessarily lie in $\{0, 1\}$ as it would equal 7 if all inputs were one.

Multiplication (i.e. an AND) is performed by executing the linear function $(1, 1)$ followed by a PBS with the negacyclic function $f_\wedge$ which maps the bit sequence $0x0$ to x.

Thus, in the case, we have $\mathcal{A} = \{ (1), (0), (1, 1), (1, 1, 1), (1, 1, 1, 1) \}$ and so $\lambda = 2$.

$P = 2^5$ and Integers Modulo Four: We can perform arithmetic modulo four, by taking a plaintext space of five bits, encoding inputs and outputs (modulo four) using the bit sequence $000xy$. The extra two bits of "message" space, before the zero bit used for padding in PBS operations, can be used for carry bits during our linear operations. Thus we have the admissible linear operations

$$\mathcal{A} = \{ (a_{1,1}), (a_{2,1}, a_{2,2}), \ldots, (a_{5,1}, \ldots, a_{5,5}) \}$$

$a_{i,j} \geq 0$ and for $\sum_{j=1}^{i} a_{i,j} \leq 5$ for $i = 1, 2, 3, 4, 5$. Thus $\lambda = 5$. The PBS operation following one of these admissible operations being for the function $f_{\mathrm{mod}\,4}$ which maps $x \longrightarrow x \pmod 4$ for values $0 \leq x \leq 16$.

Multiplication of integers modulo four is obtained using the combination of the linear map $\boldsymbol{\alpha} = (4, 1)$ followed by the PBS defined by the negacyclic function $f_\times$ given by Table 6

Input	

Input	

Input	Represents	Output	Input	Represents	Output
0	(0,0)	0	8	(2,0)	0
1	(0,1)	0	9	(2,1)	2
2	(0,2)	0	10	(2,2)	0
3	(0,3)	0	11	(2,3)	2
4	(1,0)	0	12	(3,0)	0
5	(1,1)	1	13	(3,1)	3
6	(1,2)	2	14	(3,2)	2
7	(1,3)	3	15	(3,3)	1

Table 6: Description of the function f_x.

5.6.3.14 Correctness: To enable homomorphic operations to proceed, with negligible probability of error *err*, we require the following two equations to be satisfied

$$|\mu_{PKE}| + c_{err,1} \cdot \sqrt{\sigma_{PKE}^2 + \sigma_{DS}^2}$$

$$\leq \begin{cases} c_{err,1} \cdot \sqrt{\sigma_{BS}^2 + FFTNoise(Q, \ell, q, N, \beta_{bk}, \nu_{bk}) + \sigma_{KS}^2} & \text{if } type = LWE, \\ c_{err,1} \cdot \sqrt{\sigma_{BS}^2 + FFTNoise(Q, \ell, q, N, \beta_{bk}, \nu_{bk})} & \text{if } type = F\text{-}GLWE; \end{cases} \quad (19)$$

$$\frac{\Delta}{2} > \begin{cases} c_{err,1} \cdot \sqrt{\lambda^2 \cdot (\sigma_{BS}^2 + FFTNoise(Q, \ell, w, N, \beta_{bk}, \nu_{bk}) + \sigma_{KS}^2) + \sigma_{MS}^2} & \text{if } type = LWE, \\ c_{err,1} \cdot \sqrt{\lambda^2 \cdot (\sigma_{BS}^2 + FFTNoise(Q, \ell, w, N, \beta_{bk}, \nu_{bk})) + \sigma_{KS}^2 + \sigma_{MS}^2} & \text{if } type = F\text{-}GLWE. \end{cases} \quad (20)$$

where σ_{PKE} is from equation (13), σ_{DS} is from (11), σ_{MS} is from equation (16), σ_{KS} is from equation (14), and σ_{BS} is from equation (17).

The first equation here (equation 19) is to ensure a fresh ciphertext has small enough noise standard deviation in order to be able to be further processed, i.e. the expected noise value after the application of *TFHE.Enc* followed by *TFHE.DimensionSwitch* is less than the expected noise value which is output by the full PBS operation.

The second equation (equation 20) is to ensure the PBS_f operation does not abort. Notice the difference in the position of the σ_{KS}^2 term depending on the value of *type*, this is related to the fact as to whether the key switching operation occurs before the admissible linear operations or after the admissible linear operations, i.e. the order of the execution of *TFHE.BootStrap* and *TFHE.KeySwitch* in Figure 27.

For our threshold protocol to work when we have $\overline{flag} = true$ then we also require (assuming a threshold decryption operation follows a PBS operation and not an application of a linear function $h \in \mathcal{A}$) that,

$$c_{err,1} \cdot \sqrt{\sigma_{BS}^2 + FFTNoise(Q, \ell, w, N, \beta_{bk}, \nu_{bk}) + \sigma_{KS}^2 + \overline{\sigma_{MS}}^2} \quad < \quad \frac{\Delta}{2}, \quad (21)$$

$$B_{SwitchSquash} = c_{err,1} \cdot \sqrt{\overline{\sigma_{BS}}^2 + FFTNoise(\overline{Q}, \ell, \overline{w}, \overline{N}, \overline{\beta_{bk}}, \overline{\nu_{bk}})} \quad \leq \quad 2^{70}. \quad (22)$$

The first equation here is to ensure that the *TFHE.SwitchSquash* operation does not abort, whilst the second equation is to ensure that we have enough gap after a *TFHE.SwitchSquash* operation to apply noise–flooding in our threshold decryption operation. If equation (22) is satisfied then we have the equation

$$2 \cdot nSmallBnd \cdot 2^{stat} \cdot B_{SwitchSquash} < \frac{\overline{\Delta}}{2}$$

for $\overline{\Delta} = \overline{Q}/P$ with $\overline{Q} = 2^{128}$, $P \leq 2^5$, $nSmallBnd \leq 10,000$ and $stat = 40$. It is this last equation which will imply correctness of our threshold decryption operation.

5.6.3.15   Security: To ensure security of TFHE we need to assume, just as with BGV and BFV, an additional hardness assumption. Recall the TFHE key generation algorithm in Figure 17 takes the under–lying LWE secret key **s**, and encrypts each bit via the GGSW scheme under the key $(\mathbf{s}_0, \ldots, \mathbf{s}_{w-1})$, and (again under the GGSW scheme) under the key $(\overline{\mathbf{s}}_0, \ldots, \overline{\mathbf{s}}_{\overline{w}-1})$. These two sets of encryptions form the standard and *TFHE.SwitchSquash* bootstrapping keys BK and $\overline{BK}$. In addition the keys $\mathbf{s}_i$ are encrypted under the key **s** using the *Lev* encryption scheme, to form the key–switching keys KSK and $PKSK$. Thus we have a rather intertwined form of circular security.

Thus we need to make the circular security assumption that giving out such encryptions does not weaken the security of the underlying LWE problem. We refer to this as the TFHE–Circular–Security assumption.

Definition 6 (TFHE–Circular–Security Assumption)**.** The Ring–LWE problem given by a sample of the form

$$\mathfrak{pf}_a \leftarrow \mathbf{R}_Q, \quad \mathfrak{pf}_b \leftarrow \mathfrak{pf}_a \odot \overleftrightarrow{\mathbf{s}} + \mathbf{e} \in \mathbf{R}_Q$$

for $\mathbf{e} \leftarrow TUniform(\hat{\ell}, -2^{b_i}, 2^{b_i})$, for a fixed value $\mathbf{s} \leftarrow \{0, 1\}^{\hat{\ell}}$, is still hard in the presence of the KSK, $PKSK$, BK, and $\overline{BK}$ samples generated in Figure 17.

Assuming this we have the following theorem, which is standard

Theorem 2. *The TFHE scheme is an IND–CPA Somewhat Homomorphic Encryption scheme assuming the associated Ring–LWE problem, LWE problems and GLWE problems are hard, and the TFHE–Circular–Security assumption holds.*

This circular security assumption cannot be removed in TFHE as it is needed to enable bootstrapping, and hence evaluation of functions of arbitrary depth. To ensure concrete security we simply need to select the values $b_\ell, b_{w \cdot N}$ and $b_{\overline{w} \cdot \overline{N}}$ appropriately from the values in Table 24.

5.6.3.16   Parameters: Then the question arises as to what parameters can we utilize which ensures that the associated problems are indeed hard, and we can at the same time evaluate functions homo–morphically for a given plaintext size P. We need to find parameters which satisfy the equations (19)–(22). Recall, following Design Decision 10, we assume a maximum value of $nSmallBnd = 10,000$ in deriving the parameters for the *TFHE.SwitchSquash* operation, i.e. $\overline{w}, \overline{N}, \overline{v_{bk}}$ and $\overline{\beta_{bk}}$.

There are a plethora of possible choices, but to enable as efficient an implementation as possible we prioritize the values (w, N, v_{bk}) which minimize the expression

$$cost_{ks} + cost_{bs} = (2 \cdot N \cdot w \cdot v_{ksk})) + \left(2.7 \cdot (w+1)^2 \cdot v_{bk} \cdot N + (w+1) \cdot (v_{bk}+1) \cdot N \cdot (3.2 + \log_2 N)\right).$$

This is a rough approximation to the cost of executing the *PBS* operation, see [Tap23]. We also try to minimize the same expression when (w, N, v_{bk}) is replaced by $(\overline{w}, \overline{N}, \overline{v_{bk}})$. Note that, our parameters supporting threshold operations are the same as those for non–threshold operations (unlike in the case of BGV and BFV).

As in our examples above, for admissible linear function, we examine the plaintext spaces $P = 8$ and $P = 32$, the first with $\lambda = 2$ and the second with $\lambda = 5$. This gives us (using the formulae and methods of *this* document) the parameters in Table 7, using $c_{err,1} = 13.15$ and $stat = 40$. We see that the parameters using $type = F\text{-}GLWE$ will produce a faster PBS operation, at the expense of larger ciphertexts being operated on homomorphically. For all parameter sets we assume the worst possible soundless slack of $zk\text{-}slack = \sqrt{2 \cdot \hat{\ell}}$ which can arise using the the worst zero–knowledge proof, in this respect, and the most efficient public key encryption method known[14].

Recall, as explained on pages 41 and 90 in determining these parameters we have targetted an error probability of $err = 2^{-128}$ in order to prevent IND–CPA^D attacks, using the *methods described in this document.*

---

[14]Which we note is not covered in this document, but is included in the linked Rust libraries and is explained in [BdBB$^+$25].

		This Document			TFHE–rs
	type = LWE		*type = F-GLWE*		*type = F-GLWE*
	$P = 8$	$P = 32$	$P = 8$	$P = 32$	$P = 32$
λ	2	5	2	5	5
$\hat{\ell}$	1024	2048	1024	2048	2048
ℓ	926	1004	848	926	918
w	2	1	2	1	1
N	1024	4096	1024	4096	2048
Q	2^{64}	2^{64}	2^{64}	2^{64}	2^{64}
ν_{pksk}	2	4	1	1	4
β_{pksk}	2^7	2^4	2^{15}	2^{17}	2^4
ν_{bk}	1	1	1	1	1
β_{bk}	2^{18}	2^{21}	2^{18}	2^{22}	2^{23}
ν_{ks}	2	5	2	3	4
β_{ks}	2^7	2^4	2^6	2^5	2^4
$b_{\hat{\ell}}$	42	16	42	16	17
b_{ℓ}	44	42	46	44	45
$b_{w \cdot N}$	16	0	16	0	17
$\overline{w}$	4	1	4	1	2
$\overline{N}$	1024	4096	1024	4096	2048
$\overline{Q}$	2^{128}	2^{128}	2^{128}	2^{128}	2^{128}
$\overline{\nu_{bk}}$	3	3	3	3	3
$\overline{\beta_{bk}}$	2^{24}	2^{24}	2^{24}	2^{24}	2^{24}
$b_{\overline{w} \cdot \overline{N}}$	27	27	27	27	30

Table 7: Sample parameters for TFHE.

In the last column of Table 7 we give the parameters for $P = 32$ and $\lambda = 5$ used in the Rust library **TFHE–rs**. These parameters achieve computational security parameter $sec = 128$ and failure probability $err = 2^{-128}$. They use $type = F\text{-}GLWE$, as these are the most efficient parameters. The astute reader will notice that the parameters are slightly lower than the equivalent ones derived using the methods in this document (in particular the values of N and $\overline{N}$ are half those of the equivalent ones in this document, resulting in much better performance).

To achieve such improvements one needs to implement a number of optimizations to the "basic" TFHE scheme we have described here. The main ones of these are as follows:

1. A "randomized" version of the *TFHE.ModSwitch* operation is needed, which is described in [BdBB+25, BJSW24]. The introduced "drift mitigation technique" accounts for the main improvement in param–eters in the last column of Table 7. In practice, this technique ensures that the cost of bootstrapping with a failure probability of $err = 2^{-128}$ is nearly equivalent to that of a conventional bootstrapping with $err = 2^{-64}$. The use of this technique, however, requires some encryptions of zero to be added into the public key material. Thus key generation becomes a little more complex than what is described in this document.

2. In **TFHE–rs** a four bit integer (for example) is held as two ciphertexts which encrypt the plaintext values $000b_3b_2$ and $000b_1b_0$. The three zero bits in each ciphertext correspond to the padding bit (needed for the PBS operation) and two bits of carry space (to enable a limited amount of efficient linear operations as described above. However to encrypt the four bit integer, the four bits are packed into one ciphertext which encrypts $0b_3b_2b_1b_0$. The associated ciphertext is then accom–panied with a zero–knowledge proof of correct encryption (such as those described in Section 7.6). On receipt of a ciphertext, the zero–knowledge proof is checked, then a *TFHE.DimensionSwitch*

is performed to a ciphertext **with** $type = LWE$, then two *TFHE.BootStrap*'s are performed to un–pack the single ciphertext into two ciphertexts, of $type = F\text{-}GLWE$, encrypting the plaintext values $000b_3b_2$ and $000b_1b_0$. This more complex procedure, than that described in this document, allows a reduction in the amount of zero–knowledge proofs needed during encryption. However, it means the ν_{pksk} and β_{pksk} values in the last column of Table 7 correspond to a keyswitch to a ciphertext of $type = LWE$ and not $type = GLWE$ as one would normally expect.

3. Finally, a more elaborate method to optimize for the "best" parameters is used in order to settle on the final choice of parameters, see [Tap23] for further details.

When deploying a protocol based on TFHE, one typically incorporates these optimizations and, in practice, uses the parameters from the last column. Recall we are mainly interested in describing the threshold protocols related to FHE, and such optimizations do not really affect our description of these protocols, thus we do not go into further detail in this document.

6   System model

In this section we summarize some of the more important system assumptions in one place. Some of these assumptions and decisions have been discussed earlier.

6.1   Threshold Profiles

NIST define a number of threshold profiles, for when $\mathfrak{t} < \mathfrak{n}/3$; in particular *nSfH*, *nMfH*, *nLfH* and *nEfH*. The S, M, L and E here refer to a small, medium, large or enormous number of players, and the *H* refers to the super–honest majority of $\mathfrak{t} < \mathfrak{n}/3$. We require a slightly more nuanced description, of the different profiles. So we add some additional profiles into the mix, which we list in Table 8. The main two distinctions are that our protocol choices are different depending on the size of $\binom{\mathfrak{n}}{\mathfrak{t}}$, and on whether we assume $\mathfrak{t} < \mathfrak{n}/3$ or $\mathfrak{t} < \mathfrak{n}/4$. To ease exposition we will divide the protocol profiles we will use into "small" and "large" families. Which we denote by

$$nSmall = \{nSfH, nmfH\},$$
$$nLarge = \{nMfH^{+}, nLfH^{+}, nEfH^{+}\}.$$

We shall refer to the four NIST profiles for $\mathfrak{t} < \mathfrak{n}/3$ as the *NIST* threshold profiles, i.e.

$$NIST = \{nSfH, nMfH, nLfH, nEfH\}.$$

Category	Profile	$\mathfrak{n}$	$\mathfrak{t}$	$\binom{\mathfrak{n}}{\mathfrak{t}}$
NIST	*nSfH*	$4 \leq \mathfrak{n} \leq 8$	$\mathfrak{t} < \mathfrak{n}/3$	–
	nMfH	$9 \leq \mathfrak{n} \leq 64$	$\mathfrak{t} < \mathfrak{n}/3$	–
	nLfH	$65 \leq \mathfrak{n} \leq 1024$	$\mathfrak{t} < \mathfrak{n}/3$	–
	nEfH	$\mathfrak{n} \geq 1025$	$\mathfrak{t} < \mathfrak{n}/3$	–
nSmall	*nSfH*	$4 \leq \mathfrak{n} \leq 8$	$\mathfrak{t} < \mathfrak{n}/3$	–
	nmfH	$9 \leq \mathfrak{n} \leq 64$	$\mathfrak{t} < \mathfrak{n}/3$	$< nSmallBnd$
nLarge	*nMfH^{+}*	$9 \leq \mathfrak{n} \leq 64$	$\mathfrak{t} < \mathfrak{n}/4$	–
	nLfH^{+}	$65 \leq \mathfrak{n} \leq 1024$	$\mathfrak{t} < \mathfrak{n}/4$	–
	nEfH^{+}	$\mathfrak{n} \geq 1025$	$\mathfrak{t} < \mathfrak{n}/4$	–

Table 8: Summary of Threshold Profiles.

Our extra divisions depend on whether the value of $\binom{n}{t}$ is greater than or less than *nSmallBnd*. When less than *nSmallBnd* we obtain robust protocols for $\mathfrak{t} < \mathfrak{n}/3$ and when greater than or equal to *nSmallBnd* we obtain robust protocols for $\mathfrak{t} < \mathfrak{n}/4$. The protocols in the regime of small $\binom{n}{t}$ are based on PRSS protocols, which provide efficient VSS protocols essentially for free. See the discussion related to Design Decision 10, where we discuss the pro's and con's related to the choice of *nSmallBnd*. Recall, that in this document we assume than *nSmallBnd* has a maximum value of $10,000$.

Our protocols are described for PRSS instantiations based on the original presentation given in [CDI05]; as we are working with Shamir Sharings of degree $\mathfrak{t}$. However, by increasing the degree, and having a gap between the degree and the threshold value, one could *potentially* obtain more efficient PRSS instantiations (with less keys) using the techniques described in [BBG^{+}21, EHL^{+}23] based on covering designs. Similarly we do not utilize optimizations based on packed secret sharings.

6.2 Initialization/SetUp

We assume implicitly a protocol, which we call *SetUp*, which agrees on the set of parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$, and distributes pair–wise MAC (and potentially encryption) keys between them in order to ensure that all communication between players is over authentic channels (and potentially private channels). If we require private channel communication in any protocol we will state so explicitly. In practice these pair–wise authentic channels can be set up via TLS connections and a PKI, and thus one obtains private channels in some sense "for free" (although we will rarely use private communication). All uses of private communication in protocols below are marked in **bold**.

Along with this basic setup procedure, the MPC protocols defined in Section 7 also define protocol specific setup procedures. For example, initialization protocols for the PRSS keys (for which we provide active–with–abort and robust versions). The MPC protocols have initialization routines which also define various lists of data, or other parameters. Such protocol specific initialization routines could be merged with the authenticated channel *SetUp* procedure in any instantiation.

6.3 Cryptographic Assumptions

We have cryptographic assumptions underlying the FHE schemes which we are providing threshold variants of. The threshold protocols themselves are based mainly on information theoretic constructions, and so–called "lightweight" cryptography, i.e. the application of hash functions and pseudo–random functions. Thus all constructions are inherently post-quantum secure, at the same level of security as AES–128. For those wishing post-quantum security at a level equivalent to AES–256 various modifications to what follows would be needed, but these are mainly changes in security parameters and output sizes of hash and pseudo–random functions. In particular in moving to AES–256 level of security the underlying FHE systems would also need to have increased security parameters in any case.

The only place where we utilize pre–quantum primitives is in the ZKPoKs of correct FHE encryption which are based on vector commitments. Here we utilize pairings on elliptic curves, which are not post–quantum secure. However, the zero–knowledge property holds irrespective of the security of the pairings. We only require pairing assumptions to establish soundness. The soundness of our proof systems, in an application, may not need to hold for a long time. So it maybe argued that a pre–quantum primitive for soundness may be sufficient in applications for the near future.

6.4 Offline–Online Paradigm

As outlined in Design Decision 5 we divide our MPC protocols into a function independent offline phase and an online phase which executes the desired functionality. Indeed, as will be seen later, we have an offline phase which generated multiplication triples, on top of this is another semi–offline phase which generates shared bits, and then finally on top of this are defined the threshold decryption and key generation procedures.

For threshold profiles *nSmall* we require no offline phase to execute the threshold decryption operation, and thus the offline phase (in particular the bit–generation methodology) is only needed for the key generation procedure. These protocols work for $\mathfrak{t} < \mathfrak{n}/3$.

For threshold profiles *nLarge* we require an offline phase to generate shared random bits for both threshold decryption and threshold key generation. We only define a shared random bit generation method for q a power of two; thus threshold profile *nLarge* can only be applied to thresholdize the TFHE scheme; and not the BGV or BFV scheme. Generating shared random bits is possible theoretically in the regime where q is a product of odd primes, but the protocol is relatively complex and we decided not to include it in this document. This is captured in Design Decision 13 and Design Decision 14.

6.5   Network model

We describe our protocols in two network models, both with secure authenticated point to point chan–nels:

- the synchronous model for the preprocessing phase
- the asynchronous model with eventual message delivery for the online phase

We note that a protocol that works in the asynchronous can be instantiated with a synchronous network as the synchronous model is more stringent.

Our underlying network is generally considered to be synchronous, however, in Section 7 we outline when a specific protocol does not need to make this synchronicity assumption. Generally speaking the offline phases require a synchronicity assumption, but the online phases do not.

6.5.1   Secure Authenticated Point To Point Channels

To realize the assumption that our channels are secure and authenticated, we instantiate our point to point channels using *tonic* (a Rust library) implementation of gRPC's SSL/TLS channels, and we assume that each party has access to the TLS certificates of all the parties. The question of how these certificates are distributed is out of the scope of this report.

6.5.2   Synchronous Model

In our synchronous model, parties have access to bounded–delay channels with known upper bound. This means that a message sent by an honest party is certain to be delivered within a known delay to its recipient. In practice, this known upper bound is fixed as a configuration parameter of the protocol and should take into account the expected round–trip time between the parties (e.g. this bound should be bigger in a WAN than in a LAN setting) and the time to compute the message to be sent for the current round. As such, if no message is received after this known upper bound has expired, the sender is considered malicious by the receiver. In practice, when we assume synchronous networks, we set this upper bound to ten seconds.

We note here that in models of synchronous networks, parties must ensure that the round has com–pleted for **all** the honest parties before starting the next round. For practical reasons, our implementation does not provide such guarantees as this would imply that every round has to last exactly the given upper bound, or parties need to exchange a "proceed" message. In our implementation, parties move on to the next round as soon as they are able to (i.e. once they have received messages from all other parties), without waiting for the upper bound to expire. This allows us to have rounds much shorter than the worst case upper bound with the caveat that some parties may send their message for round $r + 1$ whilst some other parties have not yet received all their messages for round r. We include the round number as metadata in every message sent to ensure that the receiver can handle it correctly, even if sender and receiver are not perfectly synchronized, i.e. moved to different rounds.

In theory this simplification could result in attacks; where an adversary sends a message in round r to one honest party in a round, making that honest party pass to round $r + 1$. However, the adversary does not send a message to a second party for round r, until it has received the response in round $r + 1$ from the first honest party. Our protocols are designed to try and avoid such problems by relying either on guaranteed broadcast protocols, or post–event checking of data.

6.5.3   Asynchronous Model

In our asynchronous model, parties only have access to a channel that ensures eventual delivery of the messages. We use the same implementation as for the synchronous model, but with an upper bound fixed to a year. However, our protocols in the asynchronous model are able to proceed faster as they are

designed to cope with some players failing to send messages within a suitable time period. In particular, when working in the asynchronous model an honest party has to move on to the computing the next message as soon as it has received $n - t$ messages where n is the number of parties and t the corruption threshold to guarantee eventual termination. This is because, by definition, when a message is missing, there is no way to wait long enough to distinguish between the case where the message was indeed sent by an honest party and took very long to arrive or if the message was never sent by a malicious party. In practice, our realizations of the synchronous and asynchronous channels do not differ much. This is mainly because in the synchronous setting we allow parties to move on to the next round as soon as they are able to. However, the added benefit of working in the asynchronous model is that we can not mistakenly consider a party adversarial because of a slower than expected network. As such, all protocols which can be run in this asynchronous setting are run in it.

6.6 Adversary

As explained earlier, we assume a static, malicious adversary, that can corrupt up to $\mathfrak{t}$ parties, and which is fully malicious; i.e. when controlling a party it may arbitrarily deviate from the protocol. This deviation can include not sending messages when required to do so.

The assumption of static adversaries is purely for modeling purposes. We conjecture that our protocols are secure even for adaptive adversaries, since they are based mainly on information theoretic constructions. The only places where adaptivity is a problem would be where we send data privately between players, as then the standard security proofs for adaptive adversaries would seem to require a form of equivocable encryption to be used. Such private transmissions are confined to sub-procedures in the offline phases of our protocols.

In Section 7 we also provide mechanisms for ReSharing a secret. This enables a sharing to be re-randomized between a set of parties, or for a share to be transferred from one set of parties to another. Such re-sharing protocols are needed in real systems to cope with parties leaving/joining a group. In addition such ReSharing, combined with a mechanism to ensure guaranteed erasure of secret shares from a previous time period, allows a form of *proactive security* to be provided. Thus if an attacker breaks into up to t participants in one time period, this does not help him when breaking into more participants in a future time period. Thus as long as the attacker does not compromise more than t parties per time interval security is maintained.

7 Protocol description

Our protocol stack is divided into a number of layers.

- At layer zero we have methodologies and schemes which are not actually protocols, but algorithms. This includes algorithms for processing elements in Galois rings, and underlying mathematical constructs. We also assume at layer zero a *SetUp* process, which we outlined above in Section 6.2.
- At the bottom layer of the main stack of protocols (which we call layer one) are those protocols related to managing the transmission, opening and initialization of the secret sharing schemes, plus other "book-keeping" protocols.
- At the next layer (layer two) are those protocols related to setting up the offline data preparation of our main core MPC protocol. This layer comes in two variants: One where $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "small", which we denote by MPC^S, and one when $\binom{\mathfrak{n}}{\mathfrak{t}}$ is "large", which we denote by MPC^L. The MPC^S protocol works for $\mathfrak{t} < \mathfrak{n}/3$, whilst the MPC^L protocol requires us to assume that $\mathfrak{t} < \mathfrak{n}/4$.
- At the next layer (layer three) are the protocols related to the online execution of the MPC. All the protocols here are able to be run **asynchronously** (except for any calls to the offline phase needed). This layer implements in essence an arithmetic black box over the underlying Galois Ring, as well as generating the shared random bits. This layer is defined totally generically, independent of the underlying protocols at layer two. We present the bit generation protocols in this layer, as they are inherently asynchronous. One could consider them in the offline layer, or even a sub-procedure in the final fourth protocol layer if desired.
- Finally we have our protocols (on layer four) for threshold key generation, threshold decryption, and resharing of the shared secrets of the FHE schemes.
- We also provide a layer five set of protocols which allow for a user to prove in zero-knowledge that an FHE ciphertext has been created correctly.

For every layer except layer zero, we assume the existance of a session ID (*sid*) and a round ID (*rid*) in every protocol. For brevity, in this document, these are not explicitly given as the input to every protocol. The session ID matches the definition of session ID in the UC framework, namely protocols running in the same session should only accept messages that is tagged with the same session ID. The round ID serves a similar role as the sub-session ID from the UC framework. Normally, in the context of the UC framework, the sub-session ID enables concurrent composition of sub-protocols under the same *sid*. None of our protocols use concurrent composition so we can use a counter that is incremented in every round (*rid*) as the sub-session ID in our description.

In what follows we will make extensive reference to the threshold profiles discussed in Section 6.1.

We summarize the protocols in Table 9, as well as what threshold profiles families they are used in in our protocol stack, and whether they work over synchronous or asynchronous networks. In the notes column we summarize exactly what properties this protocol gives us, or what requirements it has, if used as a separate protocol.

Layer	Protocol	Threshold Profile	Sync vs Async	Notes
Zero	$BW(\mathbf{c}, e)$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$Gao(\mathbf{c}, e)$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$ErrorCorrect(q, \mathbf{c}, e)$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$Correct^{\mathbb{F}}(p, \mathbf{r})$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$Correct^{GR}(q, \mathbf{r})$	All	N/A	$e \leq \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$
	$Share(a)$	All	N/A	Any, Semi-Honest
	$OpenShare(\{\langle a \rangle_i\}_{i \in S})$	All	N/A	$\mathfrak{t} < \mathfrak{n}/2$, A-w-A
	$Commit(m)$	All	N/A	ROM
	$Verify(c, o, m)$	All	N/A	ROM

	Solve(v)	All	N/A	$q = 2^k$
	Sqrt(v, p)	All	N/A	p prime
	TreePRG.Gen($seed, d$)	All	N/A	–
	TreePRG.GenSub($seed, lab, d, i$)	All	N/A	–
	TreePRG.Punc($seed, d, T$)	All	N/A	–
	TreePRG.PuncSub(D, lab, T, i)	All	N/A	–
	TreePRG.GenPunc(D, T, d)	All	N/A	–
One	*Synch-Broadcast*(S, m)	All	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	RobustOpen($\mathcal{P}, \langle a \rangle^d$)	All	Any	$d + 2 \cdot \mathfrak{t} < \mathfrak{n}$
	RobustOpen($\mathcal{S}, \langle a \rangle^d$)	All	Any	$d + 2 \cdot \mathfrak{t} < \mathfrak{n}$
	BatchRobustOpen($(\langle x_i \rangle^d)_{i=1}^{\ell}$)	All	Sync	$d + 2 \cdot \mathfrak{t} < \mathfrak{n}$
	VSS($\mathcal{P}_k, s, \mathfrak{t}, Corrupt$)	All	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	AgreeRandom($\mathcal{S}, k$)	*nSmall*	Sync	Any, Semi–Honest
	AgreeRandom-w-Abort($\mathcal{S}, k$)	*nSmall*	Sync	Any, A–w–A
	AgreeRandom-Robust($\mathcal{S}, k, \langle r \rangle$)	*nSmall*	Any	$\mathfrak{t} < \mathfrak{n}/3$
	PRSS.Init()	*nSmall*	Sync	Any, A–w–A
	PRSS.Init($Corrupt$)	*nSmall*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	PRSS.Next()	*nSmall*	N/A	Any
	PRSS.Check($cnt, Corrupt$)	*nSmall*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	PRZS.Next()	*nSmall*	N/A	Any
	PRZS.Check($cnt, Corrupt$)	*nSmall*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	PRSS-Mask.Next($Bd, stat$)	*nSmall*	N/A	Any
	CoinFlip($Corrupt$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
Two	MPC^S.*Init*()	*nSmall*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	MPC^S.*GenTriples*($Dispute$)	*nSmall*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	MPC^S.*NextRandom*($Dispute$)	*nSmall*	N/A	Any
	ShareDispute($\mathcal{P}_i, s, d, Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	LocalSingleShare($\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$, ROM
	LocalDoubleShare($\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$, ROM
	SingleSharing.Init($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	SingleSharing.Next($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	DoubleSharing.Init($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	DoubleSharing.Next($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	MPC^L.*Init*($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
	MPC^L.*GenTriples*($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/4$
	MPC^L.*NextRandom*($Dispute$)	*nLarge*	Sync	$\mathfrak{t} < \mathfrak{n}/3$
Three	*MPC.Open*($\langle x \rangle$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	MPC.Mult($\langle x \rangle, \langle y \rangle, Corrupt$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	MPC.GenBits(v)	All(*)	Any	$\mathfrak{t} < \mathfrak{n}/3$
Four	*MPC.NewHope*(N, B)	All(*)	Any	$\mathfrak{t} < \mathfrak{n}/3$
	MPC.TUniform($N, -2^b, 2^b$)	All(*)	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BGV.Threshold-KeyGen($\ldots$)	*nSmall*	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BGV.Threshold-Dec($\mathfrak{ct}, \langle \mathfrak{sf} \rangle$)	*nSmall*	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BFV.Threshold-KeyGen($\ldots$)	*nSmall*	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BFV.Threshold-Dec($\mathfrak{ct}, \langle \mathfrak{sf} \rangle$)	*nSmall*	Any	$\mathfrak{t} < \mathfrak{n}/3$
	TFHE.Threshold-KeyGen($\ldots$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	TFHE.Threshold-Dec-1($\mathfrak{ct}, PK, \langle \mathfrak{sf} \rangle$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	XOR($\langle a \rangle, \langle b \rangle$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BitAdd($(\langle a_i \rangle)_{i=0}^{k-1}, (\langle b_i \rangle)_{i=0}^{k-1}$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	BitSum($(\langle a_i \rangle)_{i=0}^{k-1}$)	All	N/A	–
	BitDec($\langle a \rangle$)	$q = 2^k$	Any	$\mathfrak{t} < \mathfrak{n}/3$
	TFHE.Threshold-Dec-2($\mathfrak{ct}, \langle \mathfrak{sf} \rangle$)	All	Any	$\mathfrak{t} < \mathfrak{n}/3$
	ReShare($S_1, S_2, \langle \mathfrak{sf} \rangle^{S_1}$)	All	Sync	$\mathfrak{t}_1 < \mathfrak{n}_1/3, \mathfrak{t}_2 < \mathfrak{n}_2/3$
Five	*CRS-Gen*($sec, \bar{q}, \bar{d}, \bar{B}$)	All	Sync	$\mathfrak{t} < \mathfrak{n}$
	CRS-Gen.Init($sec, \bar{q}, \bar{d}, \bar{B}$)	All	Sync	$\mathfrak{t} < \mathfrak{n}$
	CRS-Gen.Update(pp_{j-1})	All	Sync	$\mathfrak{t} < \mathfrak{n}$
	CRS-Gen.Output()	All	Sync	$\mathfrak{t} < \mathfrak{n}$
	VC-Prove-1($\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b}$)	N/A	N/A	ROM
	VC-Verify-1($\mathbf{p}, (\mathbf{A}, \mathbf{s}), prf$)	N/A	N/A	ROM

VC-Prove-2(**p**, (**A**, **s**), **b**)	N/A	N/A	ROM
VC-Verify-2(**p**, (**A**, **s**), *prf*)	N/A	N/A	ROM
MPCitHead-Prove-1((**A**, **s**), **b**)	N/A	N/A	ROM
MPCitHead-Verify-1((**A**, **s**), *prf*)	N/A	N/A	ROM
MPCitHead-Prove-2((**A**, **s**), **b**)	N/A	N/A	ROM
MPCitHead-Verify-2((**A**, **s**), *prf*)	N/A	N/A	ROM
XOF-Share(*x*, *XOF*)	N/A	N/A	–
MPCitHead-Prove-3((**A**, **s**), **b**)	N/A	N/A	ROM
MPCitHead-Verify-3((**A**, **s**), *prf*)	N/A	N/A	ROM

Table 9: Summary of Protocols. All protocols are robust, unless marked A–w–W or Semi–Honest. An All(*) means all except *nLarge* when $q = p_1 \cdots p_k$.

7.1 Layer Zero

7.1.1 Galois Rings

Throughout this document we let q denote the modulus for our secret sharing scheme. In applications this is related to a ciphertext modulus of the underlying FHE scheme. For example in TFHE it may be either Q or $\overline{Q}$, in BGV/BFV it can be the modulus at the lowest level, i.e. Q_1, or the highest level $Q = Q_1 \cdots Q_L$.

The secret sharing modulus q will either be a prime power $q = p^k$, or will be a product of distinct primes $q = p_1 \cdots p_k$. In the latter case we let p denote the **smallest** prime dividing q. When $q = p^k$ it will be the case, in our applications, that p is very small (usually $p = 2$). When $q = p_1 \cdots p_k$ it will be the case, in our applications, that p is relatively large (over 20 bits). These last facts needs to be kept in mind when reading this section.

In addition, throughout this document, we let $F(X)$ denote a monic polynomial in $\mathbb{Z}[X]$ which is irreducible modulo all primes dividing q, and has degree $\mathfrak{d}$. We can think of F as also being defined over $\mathbb{Z}_p$, or over $\mathbb{Z}/(q)$. That $F(X)$ is irreducible modulo p means that the p–adic ring of integers

$$\mathbb{Z}_p[\theta] = \mathbb{Z}_p[X]/F(X)$$

is an unramified extension of $\mathbb{Z}_p$ of degree $\mathfrak{d}$. See [Cas86] for a thorough treatment of ring extensions of $\mathbb{Z}_p$ and further results.

Recall, by Parameter Choice 3, the degree $\mathfrak{d}$ and the prime p are related to the number of players $\mathfrak{n}$ we will be able to support in our protocols by the equation

$$p^{\mathfrak{d}} > \mathfrak{n}.$$

If this equation is not satisfied, then $\mathfrak{d}$ will need to be increased for the application.

When $q = 2^k$, and since are only going to be interested in values of n less than 255, to fix the representations in this document we select the irreducible polynomial F of degree $\mathfrak{d}$ when q is a power of two according to Table 10. When $q = p_1 \cdots p_k$, the value of p will already be very large in our application, and so, in this case, one can take $\mathfrak{d} = 1$ and $F(X) = X$.

n	$\mathfrak{d}$	Polynomial
$4 \leq n \leq 7$	3	$X^3 + X + 1$
$8 \leq n \leq 15$	4	$X^4 + X + 1$
$16 \leq n \leq 31$	5	$X^5 + X^2 + 1$
$32 \leq n \leq 63$	6	$X^6 + X + 1$
$64 \leq n \leq 127$	7	$X^7 + X + 1$
$128 \leq n \leq 255$	8	$X^8 + X^4 + X^3 + X + 1$

Table 10: Table of irreducible polynomials $F(X)$ to be used when $q = 2^k$.

We define the Galois ring

$$GR(q, F) = (\mathbb{Z}/(q))[X]/F(X).$$

Just as all finite fields of characteristic p and degree $\mathfrak{d}$ are isomorphic, it turns out that all Galois rings of degree $\mathfrak{d}$ over $\mathbb{Z}/(q)$ are isomorphic. Thus one could simply talk about $GR(q, \mathfrak{d})$ and ignore the specific value of F. However, as we aim to be concrete, and enable implementers to implement algorithms, we suppose that a specific F has been chosen and that the ring $GR(q, \mathfrak{d})$ is explicitly represented as a polynomial ring with arithmetic performed modulo q and F.

When q is a prime power, i.e. $q = p^k$, we can alternatively think of $GR(q, F)$ as the ring $\mathbb{Z}_p[\theta]$ but held with only k digits of p–adic precision. Indeed, in that case, we have the ring isomorphism

$$GR(q, F) \cong \mathbb{Z}_p[\theta]/(p^k).$$

106

Since $\mathbb{Z}_p[\theta]$ is an unramified extension we have that $GR(q, F)$ contains a unique maximal ideal (p), i.e. the ideal consisting of all multiples of p. In particular

$$GR(q, F)/(p) \cong \mathbb{F}_{p^{\mathfrak{d}}},$$

where $\mathbb{F}_{p^{\mathfrak{d}}}$ is the finite field of characteristic p and degree $\mathfrak{d}$. We let this isomorphism be denoted by

$$\pi : GR(q, F) \longrightarrow \mathbb{F}_{p^{\mathfrak{d}}}, \qquad (23)$$

which we refer to as the "reduction modulo p map". One can think of $\mathbb{F}_{p^{\mathfrak{d}}} = GR(p, F)$ as being contained (as a set, and not as a field) inside $GR(q, F)$.

When $q = p_1 \cdots p_k$ the Galois ring $GR(q, F)$ is equal to $\mathbb{Z}/(q)$, since we pick $F(X) = X$ in this case. In this case we can also consider

$$GR(p_i, F) = \{-\lceil p_i/2 \rceil + 1, \ldots, \lfloor p_i/2 \rfloor\} \cong \mathbb{F}_{p_i}$$

as being contained in $GR(q, F) = \mathbb{Z}/(q)$ as a set. In this case we have a map $\pi_{p_i} : GR(q, F) \longrightarrow \mathbb{F}_{p_i}$ for all prime divisors p_i of q.

At various points we need to convert a Galois Ring element to a bit/byte string. This is done via the function *GRencode*(α). An element in $GR(q, F)$ can be considered as a vector of $\mathfrak{d}$ elements in $\mathbb{Z}/(q)$, i.e.

$$\alpha = a_0 + a_1 \cdot X + \ldots + a_{\mathfrak{d}-1} \cdot X^{\mathfrak{d}}$$

with $a_i \in \mathbb{Z}/(q)$. Each element in $\mathbb{Z}/(q)$ we encode as its byte representation, consisting of exactly $\lceil q/8 \rceil$ bytes. We then implement *GRencode*(α) as

$$GRencode(\alpha) = a_0 \| a_1 \| \ldots \| a_{\mathfrak{d}-1},$$

which is a byte string of length $\mathfrak{d} \cdot \lceil q/8 \rceil$.

Note a major problem, which we need to be careful to work around, is that $GR(q, F)$ has zero–divisors, and is hence not an integral domain. One result of this is that we need to introduce the notion of an exceptional sequence (note in our work the order of the elements is important thus we talk about exceptional sequences and not exceptional sets).

Definition 7 (Exceptional Sequence)**.** Let R denote a general commutative ring, with identity. Let $\mathcal{E} = (\alpha_1, \ldots, \alpha_n) \in R^{\mathfrak{n}}$ denote a sequence of n elements in R. We say that the sequence is an **exceptional sequence** if for all $i, j \in [1, \ldots, n]$ with $i \neq j$ we have that $\alpha_i - \alpha_j \in R^*$.

Definition 8 (Lenstra Constant)**.** The **Lenstra constant** of a commutative ring, with identity, is the maximum length of an exceptional sequence in R.

The Lenstra constant of our Galois rings $GR(q, F)$, when either $q = p^k$ or $q = p_1 \cdots p_k$, is equal to $p^{\mathfrak{d}}$, and we can take as an exceptional sequence of maximal length the subset $GR(p, F) \cong \mathbb{F}_{p^{\mathfrak{d}}}$, see Lemma 11 later.

Exceptional sequences are important to us, because the process of evaluating a polynomial at the points in an exceptional sequence is invertible. In other words given the evaluation of a polynomial of degree less than $\mathfrak{n}$, at the points in an exceptional sequence of size at least $\mathfrak{n}$, we can interpolate the points to recover the original polynomial. To see this one can just consider standard Lagrange interpolation.

Theorem 3 (Lagrange Interpolation)**.** *Let* $\mathcal{E} = (\alpha_1, \ldots, \alpha_{\mathfrak{n}})$ *denote an exceptional sequence in a commutative ring* R *with identity. Define*

$$L_j(Z) = \prod_{i \neq j} (Z - \alpha_j)$$

then for all polynomials $f(Z)$ of degree at most $\mathfrak{n}$ with coefficients in R we have

$$f(Z) = \sum_{i=1}^{\mathfrak{n}} \frac{f(\alpha_i) \cdot L_i(Z)}{L_i(\alpha_i)}.$$

Proof. The theorem follows from the usual proof of Lagrange interpolation; with the fact that we evaluate at an exceptional sequence being used to ensure that one can divide by $L_i(\alpha_i)$ in the formula. $\qquad\square$

To ease notation later we write

$$\delta_i(Z) = \frac{L_i(Z)}{L_i(\alpha_i)}. \tag{24}$$

These polynomials have the properties:

1. $\delta_i(\alpha_i) = 1.$
2. $\delta_i(\alpha_j) = 0,$ if $i \neq j.$
3. $\deg \delta_i(X) = \mathfrak{n} - 1.$

If $I \subseteq \{1, \ldots, \mathfrak{n}\}$ is a subset of indices of the exceptional set then we also define, for later, the Lagrange polynomials

$$L_i^I(Z) = \prod_{j \in I \setminus \{i\}} (Z - \alpha_j)$$

for $i \in I$. This allows us to interpolate polynomials of degree less than $|I|$ from evaluations at elements in I via

$$f(Z) = \sum_{i \in I} f(\alpha_i) \cdot \delta_i^I(Z)$$

where

$$\delta_i^I(Z) = \frac{L_i^I(Z)}{L_i^I(\alpha_i)}. \tag{25}$$

For later use we define the Vandermonde matrix $M_{r,c}$ of dimension $r \times c$, for $r \geq c$, with entries coming from an exceptional sequence $(\alpha_1, \ldots, \alpha_r)$ of size r, i.e.

$$M_{r,c} = \begin{pmatrix} 1 & \alpha_1 & \ldots & \alpha_1^{c-1} \\ 1 & \alpha_2 & \ldots & \alpha_2^{c-1} \\ \vdots & \vdots & & \vdots \\ 1 & \alpha_r & \ldots & \alpha_r^{c-1} \end{pmatrix}$$

The matrix $M_{r,c}$ is super-invertible, as it is a Vandermonde matrix generated by elements from an exceptional sequence. Being super-invertible means that any subset of c rows and columns of $M_{r,c}$ forms an invertible $c \times c$ matrix. That the matrix is super-invertible follows from the fact that $\alpha_i - \alpha_j$ is invertible in the ring.

We will make use of the Schwartz-Zippel Lemma over Galois Rings to bound the probability of failure in some of our protocols. This is given by

Lemma 10 (Schwartz-Zippel Lemma over Galois Rings)**.** *Let $G(X_1, \ldots, X_n)$ denote a multinomial with coefficients in $GR(q, F)$ of total degree $d \geq 1$. Let $r_1, \ldots, r_n$ be selected uniformly at random from an exceptional set S, then*

$$\Pr[\, G(r_1, \ldots, r_n) = 0 \,] \leq \frac{d}{|S|}$$

In our application we will use this for multinomials of degree one, and so the probability will be bounded by $1/|S|$. By selecting $S = GR(p, F)$ we obtain a bound of $1/p^{\mathfrak{d}}$. In many applications this will not be enough to ensure an exponentially small failure probability. In such situations we apply the above Lemma multiple times in order to obtain any desired small probability.

7.1.2 Reed–Solomon Codes over Galois Rings

In this work we will focus on "traditional view" Reed–Solomon codes, as they map directly onto Shamir Secret Sharing considered later. There is another view of Reed–Solomon codes (over fields), called the BCH view (which is the way algorithms are often represented in coding theory presentations). For the finite field case the two views are equivalent when one takes, as an exceptional sequence the set $\{\alpha_i\}$ to be a complete set of $\mathfrak{n}$-roots of unity. This however requires such $\mathfrak{n}$-roots of unity to exist in the first place, which can be a problem in our applications. Thus we focus on general "traditional view" Reed–Solomon codes in this work. Traditional view Reed–Solomon codes can be seen as a special case of what are called *generalized Reed–Solomon* codes in [Hal15][Chapter 5].

Definition 9 (Reed–Solomon Code). For $0 \leq \mathfrak{b} \leq \mathfrak{n}$ we define the Reed–Solomon code over the ring R, with exceptional set $\mathcal{E}$ of size $\mathfrak{n}$, as the set

$$RS_{\mathfrak{n},\mathfrak{b}} = \left\{(f(\alpha_1), \ldots, f(\alpha_{\mathfrak{n}})) \; : \; f \in R[Z], \; deg(f) < \mathfrak{b}\right\}.$$

The code maps polynomials of degree less than $\mathfrak{b}$ into a vector of $\mathfrak{n}$ ring elements. This code has minimal distance $d_{min} = \mathfrak{n} - \mathfrak{b} + 1$, and so general coding theory tells us that we can detect $\mathfrak{n} - \mathfrak{b}$ errors and correct $\lfloor (\mathfrak{n} - \mathfrak{b})/2 \rfloor$ errors. We are interested here in how this is *explicitly* performed.

Our Reed–Solomon code is generated by the Vandermonde matrix $M_{n,\mathfrak{b}}$

$$\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{b}} \iff \mathbf{c}^\top = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{\mathfrak{b}-1} \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{\mathfrak{b}-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha_{\mathfrak{n}} & \alpha_{\mathfrak{n}}^2 & \ldots & \alpha_{\mathfrak{n}}^{\mathfrak{b}-1} \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{\mathfrak{b}-1} \end{pmatrix} = M_{\mathfrak{n},\mathfrak{b}} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{\mathfrak{b}-1} \end{pmatrix}.$$

That decoding is possible (in the presence of no errors) follows from the fact that the matrix $M_{n,\mathfrak{b}}$ is super-invertible (or equivalently from the Lagrange interpolation formulae, given in Theorem 3 above).

Another characterization of $RS_{\mathfrak{n},\mathfrak{b}}$ is the following, for $r = \mathfrak{n} - \mathfrak{b}$,

$$\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{b}} \iff \sum_{i=1}^{\mathfrak{n}} c_i \cdot \frac{\alpha_i^j}{L_i(\alpha_i)} = 0 \quad \forall j \text{ such that } 0 \leq j \leq r-1, \tag{26}$$

where $L_i(Z)$ are the Lagrange polynomials from before. This follows from standard arguments using the dual Generalized Reed–Solomon code, see [Hal15][Chapter 5], which are easily seen to generalize to the Galois Ring case.

We define the **syndrome polynomial** of the received vector $\mathbf{c}$ by

$$S_{\mathbf{c}}(Z) \equiv \sum_{i=1}^{\mathfrak{n}} \frac{c_i}{L_i(\alpha_i) \cdot (1 - \alpha_i \cdot Z)} \quad (\text{mod } Z^r). \tag{27}$$

To detect errors in the code we simply need to compute the r linear equations, and see if they are all zero or not. This means we can *detect* up to the potentially $r = \mathfrak{n} - \mathfrak{b}$ errors. These r linear equations are all zero, if and only if, the above syndrome polynomial is identically zero, i.e. we have $\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{b}}$ if and only if $S_{\mathbf{c}}(Z) = 0$. For the proof of this see Lemma 12 later.

Assume a vector $\mathbf{c}$ is received, which is a noisy version of a Reed–Solomon code word. The vector has s known erasure positions (marked with a $\perp$ symbol), and at most e unknown error positions, where $2 \cdot e + s \leq \mathfrak{n} - \mathfrak{b}$. We provide two methods for error correction in such a situation. The first, using either Berlekamp–Welch or Gao as a subroutine, takes as input the received vector and outputs the corrected codeword. The second method, called syndrome decoding, which we only consider in the case where $s = 0$, takes as input the syndrome polynomial and outputs the error locations and the errors in those locations. The reason for having two methods, is that the first method will be used when error–correcting

secret shares received from different players, whilst the second will be used to correct a sharing of a sharing (which is a sub–protocol needed in some protocols such as the BGW MPC protocol [BGW88], and for our resharing protocol later).

7.1.2.1 Error Correction over Finite Fields via Berlekamp–Welch: Our decoding algorithm will output either the polynomial which generates the corrected codeword, or it outputs the symbol $\perp$, which indicates that the input has more than e errors. It works by counting the number of erasure positions s, and then applies the standard Berlekamp–Welch algorithm over the $n - s$ remaining values.

Berlekamp–Welch over $\mathbb{F}_{p^{\mathfrak{b}}}$

$BW(\mathbf{c}, e)$:
1. Let s denote the number of values with $c_i = \perp$.
2. First check (if $s = 0$) whether $\mathbf{c} \in RS_{\mathfrak{n}, \mathfrak{b}}$ (by computing syndromes), if it is return the underling $f(Z) \in \mathbb{F}_{p^{\mathfrak{b}}}[Z]$ by Lagrange interpolation.
   This step could be done for $s > 0$ by using different syndromes, but we ignore this complication here.
3. Let $\mathcal{A} = \{\alpha_i\}_{c_i \neq \perp}$.
4. For $\ell \in [e, \dots, 0]$ do:
   (a) Define a polynomial $Q(Z)$ of degree less $\mathfrak{b} + \ell - 1$, with unknown coefficients.
   (b) Define a monic polynomial $Q(Z)$ of degree ℓ, with unknown coefficients.
   (c) Solve the system of linear equations, over $\mathbb{F}_{p^{\mathfrak{b}}}$,

   $$c_i \cdot E(\alpha_i) - Q(\alpha_i) = 0 \text{ for } \alpha_i \in \mathcal{A},$$

   Note we have at least $n - s$ equations, in $\mathfrak{b} + \ell + \ell = \mathfrak{b} + 2 \cdot \ell$ unknowns; and by definition $\mathfrak{b} + 2 \cdot \ell \leq \mathfrak{b} + 2 \cdot e \leq n - s$.
   (d) If a solution exists and $q(Z) \pmod{E(Z)} \not\equiv 0$ return $f(Z) = Q(Z)/E(Z)$.
5. Return $\perp$.

Figure 29: Reed–Solomon Decoding using Berlekamp–Welch for the Finite Field $\mathbb{F}_{p^{\mathfrak{b}}}$.

In Figure 29 we present the "standard" Berlekamp–Welch algorithm, [BW86], for when we are working over finite fields, i.e. $GR(p, F) = \mathbb{F}_{p^{\mathfrak{b}}}$. The input is a vector, $\mathbf{c}$, which can contain symbols $\perp$, indicating that this location is known to be in error already; we treat such errors as erasures. In this algorithm we map our exceptional sequence to its reduction modulo p, in order to have $\alpha_i \in \mathbb{F}_{p^{\mathfrak{b}}}$. The algorithms works by finding a monic polynomial $E(Z)$, called the error locator polynomial, which is such that $E(\alpha_i) = 0$ in the positions for which $c_i \neq f(\alpha_i)$ for the underlying polynomial f which should define the valid codeword. In the main loop we assume there are exactly ℓ such errors (not including the erasures). Thus $E(Z)$ has degree ℓ. We define $Q(Z) = f(Z) \cdot E(Z)$ which will then have degree $(\mathfrak{b} - 1) + \ell$. When $c_i = \perp$ we already know there is an error, so we can define $E(\alpha_i) = 0$. When $c_i \neq \perp$ then we have the equation

$$Q(\alpha_i) = f(\alpha_i) \cdot E(\alpha_i) = c_i \cdot E(\alpha_i).$$

Thus we have $n - s$ linear equations in the $(\mathfrak{b} + \ell) + \ell$ unknown coefficients. If there is no solution then this says that the received vector does not contain ℓ errors. If there is a solution then by dividing the resulting $Q(Z)$ by $E(Z)$ (if possible) we can determine the underlying polynomial f.

7.1.2.2 Error Correction over Finite Fields via Gao: An alternative to the Berlekamp–Welch algorithm is Gao decoding [Gao03], which also works over a finite field $GR(p, F) = \mathbb{F}_{p^{\mathfrak{b}}}$ and which we present in Figure 30. Gao decoding is based on the application of the extended Euclidean algorithm to the interpolated received vector $\mathbf{c}$ and the polynomial that contains the α_i values as roots. If $\mathbf{c}$ contains less than e errors, the result of Gao is the polynomial $f(x)$, which generates the valid codeword and the

polynomial $v(x)$, which encodes the error locations at its roots. Otherwise the algorithm returns $\perp$ to indicate that the received vector could not be decoded successfully.

<div style="border:1px solid black; padding:1em;">

Gao Decoding over $\mathbb{F}_{p^{\mathfrak{b}}}$

Gao($\mathbf{c}, e$):
1. Let s denote the number of values with $c_i = \perp$.
2. Let $\mathcal{A} = \{\alpha_i\}_{c_i \neq \perp}$.
3. Precompute the polynomial

$$g_0(Z) = \prod_{\alpha_i \in \mathcal{A}} (Z - \alpha_i) \in \mathbb{F}_{p^{\mathfrak{b}}}[Z].$$

4. Via Lagrange interpolation find the unique polynomial $g_1(Z) \in \mathbb{F}_{p^{\mathfrak{b}}}[Z]$ with $deg(g_1(Z)) \leq n - s - 1$ such that, for all $\alpha_i \in \mathcal{A}$,
$$g_1(\alpha_i) = c_i.$$

5. Apply the extended Euclidean algorithm to $g_0(Z)$ and $g_1(Z)$. Stop when $deg(g(Z)) < (\mathfrak{n} - s + \mathfrak{d})/2$. This gives us

$$g(Z) = u(Z) \cdot g_0(Z) + v(Z) \cdot g_1(Z).$$

6. If $deg(v(Z)) > e$, then return $\perp$.
7. Divide $g(Z)$ by $v(Z)$, such that

$$g(Z) = f(Z) \cdot v(Z) + r(Z)$$

where $deg(r(Z)) < deg(v(Z))$.
8. If $r(Z) = 0$ and $deg(f(Z)) < \mathfrak{d}$, then return $f(Z)$.
9. Return $\perp$.

</div>

Figure 30: Reed–Solomon Decoding using Gao Decoding for the Finite Field $\mathbb{F}_{p^{\mathfrak{b}}}$.

7.1.2.3 Error Correction over Rings via Berlekamp–Welch/Gao: Our two cases of $q = p^k$ and $q = p_1 \cdots p_k$ require two distinct methods to apply error correction. See Figure 31 for the combined algorithm. Note, both algorithms can be applied to a non–full set of values by restricting the size of the exceptional sequence used to the values corresponding to those in the received vector.

The algorithm of Figure 31 uses an extension to the map π from equation (23), where we define $\pi(\perp / p^i) = \perp$ for all i and also $\pi(x / p^i) = \perp$ for all integers $x \neq 0 \pmod{p^i}$. This is done purely for ease of exposition, an implementation may decide to just set the output of π on such input values to any fixed value, and then rely on the error–correction properties of the underlying finite field error correction routine to solve any ambiguities[15].

The first case, of $q = p^k$, utilizes Hensel–lifting in order to "lift" the decoded received vector modulo p^t into a codeword modulo p^{t+1}. This lifting procedure, to a code defined over $GR(q, F)$, requires at most k calls to the sub–procedure in Figure 29 or Figure 30 for the finite field $\mathbb{F}_{p^{\mathfrak{b}}}$; see Lemma 13 later for a justification of the correctness of the method. Note, on input to the algorithm there are known to be at most e errors, and s missing values marked with $\perp$ symbols. As each new $\perp$ symbol is introduced into the vector $\mathbf{z}$ by a determined error position, this means at line 1(b)iii that the number of unknown error positions has been decreased by $s' - s$. Thus, the subsequent calls to $BW(\mathbf{z}, e + s - s')$ or $Gao(\mathbf{z}, e + s - s')$ can utilize the fact that there are less error locations to find.

The second case, of $q = p_1 \cdots p_k$, utilizes the Chinese Remainder Theorem (CRT) to lift a solution modulo p_i, for all primes p_i, into a solution modulo q. The solution modulo p_i is obtained by calling

---

[15]Although this may be slightly less efficient as then the increasing number of known erasure positions cannot be used as an optimization in the finite field subroutine.

ErrorCorrect(q, **c**, e): If e is not specified then we assume $e = \lfloor (\mathfrak{n} - s - \mathfrak{d})/2 \rfloor$, where s is the number of positions in **c** set to $\perp$.

1. If $q = p^k$ then
   (a) **y** $\leftarrow$ **c**.
   (b) For $i \in [0, \ldots, k - 1]$ do
      i. **z** $\leftarrow \pi(\textbf{y}/p^i)$.
      ii. Let s' denote the number of values with $z_i = \perp$.
      iii. Call $f_i(Z) \leftarrow BW(\textbf{z}, e + s - s')$ or $Gao(\textbf{z}, e + s - s')$ with respect to $\mathbb{F}_{p^{\mathfrak{d}}}$.
      iv. If $f_i(Z) = \perp$ then return $\perp$. In this case there are more than e errors.
      v. For $j \in [1, \ldots, n]$ set $t_j \leftarrow \sum_{l=0}^{i} f_l(\alpha_j) \cdot p^l \pmod{q}$.
      vi. **y** $\leftarrow$ **c** $-$ **t**.
   (c) Return $\sum_{l=0}^{i} f_l(Z) \cdot p^l$.
2. else
   (a) For $i \in [1, \ldots, k]$ do
      i. **z** $\leftarrow \pi_{p_i}(\textbf{c})$.
      ii. Call $f_i(Z) \leftarrow BW(\textbf{z}, e)$ or $Gao(\textbf{z}, e)$ with respect to $\mathbb{F}_{p_i}$.
      iii. If $f_i(Z) = \perp$ then return $\perp$. In this case there are more than e errors.
   (b) Return $CRT(f_1, \ldots, f_t)$.

Figure 31: Reed–Solomon Error Correction for the Galois Ring $GR(q, F)$.

Figure 29 or Figure 30 for the finite field $\mathbb{F}_{p_i}$. We let the lifting via the CRT of polynomials modulo p_i to a single polynomial modulo q be denoted by $CRT(f_1, \ldots, f_t)$.

7.1.2.4 Syndrome Decoding: Decoding on input of a syndrome for Reed–Solomon encoding is usually explcitly presented for BCH–view Reed–Solomon encoding, although syndrome decoding can be applied to any linear code. For example, this is how Wikipedia presents syndrome decoding for Reed–Solomon codes. Indeed the famous BGW paper [BGW88], on MPC, also restricts to BCH–view Reed–Solomon codes to enable syndrome decoding. Syndrome decoding will be useful later when we need to perform correction on a secret sharing of a secret sharing.

Since our traditional view Reed–Solomon codes are linear codes one does not need to adopt the BCH–view to enable syndrome decoding, as is shown explicitly in Chapter 5 of [Hal15] in the case of Reed–Solomon codes over finite fields. After presenting the method for finite fields in Figure 32, we then "lift" the method to perform syndrome decoding over the Galois ring in Figure 33. See Section 8.1.2.4 for a discussion of the correctness of the methods presented here.

Note, in this section we assume that the exceptional set consists entirely of invertible elements, i.e. we require $\alpha_i \not\equiv 0 \pmod{p}$ for all i. We also assume that the received code word contains no erasures. In our application to Shamir Secret Sharing, this will happen by default.

We now extend the above method to codes defined over the Galois Ring $GR(q, F)$. We do this much as we did before, using either Hensel lifting (when $q = p^k$) or using the Chinese Remainder Theorem (when $q = p_1 \cdots p_k$). This method is described in Figure 33.

For the case of $q = p^k$ we utilize the following notational conventions: The syndrome polynomial $S_{\textbf{c}}(Z)$ is defined in exactly the same way, and we also have the polynomials $\sigma(Z)$ and $\omega(Z)$, as above, satisfying equations (47), (48) and (49). To avoid confusion we let $\sigma^{(k)}(Z)$, $\omega^{(k)}(Z)$ denote the polynomials when considering the ring $GR(q, F)$. We derive the error vector **e** via a form of Hensel lifting. We first determine **e** modulo p, and then use this to derive **e** modulo p^2 and so on.

Note our algorithm can correct errors, when $q = p^k$, for which the error **locations** are different in $\textbf{e}^{(j)}$ and $\textbf{e}^{(j')}$, as long as the number of error locations are bounded by $\lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor$ at each p–adic level. We will not utilize this in applications, but this ability is explored in full generality in [GTLBNG21].

Syndrome Decoding over $\mathbb{F}_{p^b}$

$SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$: All these operations are performed modulo p.
1. $T \leftarrow 0, T' \leftarrow 1$.
2. $R \leftarrow Z^r, R' \leftarrow S_{\mathbf{e}}(Z)$.
3. While $deg(R) \geq r/2$
   (a) $Q \leftarrow R/R'$.
   (b) $(T, T') \leftarrow (T', T - Q \cdot T')$.
   (c) $(R, R') \leftarrow (R', R - Q \cdot R')$.
4. $\sigma(Z) \leftarrow T(Z)/T(0)$.
5. $\omega(Z) \leftarrow R(Z)/T(0)$.
6. Let B denote the indices b such that $\sigma(1/\alpha_b) = 0$.
7. For $b \in B$ define
$$e_b \leftarrow \frac{-\alpha_b \cdot L_b(\alpha_b) \cdot \omega(1/\alpha_b)}{\sigma'(1/\alpha_b)},$$
   where $\sigma'(Z)$ is the formal derivative of $\sigma(Z)$.
8. For $b \notin B$ set $e_b \leftarrow 0$.
9. Return $\mathbf{e}$.

$Correct^{\mathbb{F}}(p, \mathbf{r})$:
1. Compute the syndrome polynomial $S_{\mathbf{r}}(Z) = S_{\mathbf{e}}(Z)$ via equation (27).
2. $\mathbf{e} \leftarrow SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$.
3. $\mathbf{c} \leftarrow \mathbf{r} - \mathbf{e}$.
4. Return $\mathbf{c}$.

Figure 32: Decoding via Syndromes for Reed–Solomon Codes for the Finite Field $\mathbb{F}_{p^b}$.

7.1.3 Shamir Secret Sharing over Galois Rings

A secret sharing scheme amongst $\mathfrak{n}$ players is defined by two algorithms:
- An algorithm $Share(a)$ which outputs a share value $\langle a \rangle_i$ for each player $\mathcal{P}_i$.
- An algorithm $OpenShare(\{\langle a \rangle_i\}_{i \in S})$ which takes all the share values from a given set S and returns either the purported shared value a, or an error symbol $\perp$.

We denote the sharing of the value s by $\langle s \rangle$, and the individual shares by $\langle s \rangle_i$.

We are interested in threshold secret sharing schemes, which are defined by a parameter $\mathfrak{t} \in [1, \ldots, \mathfrak{n} - 1]$. The security requirement is that if less than or equal to $\mathfrak{t}$ number of players combine their secret shares, then they can obtain no information about the underlying shared secret, but if more than $\mathfrak{t}$ honest players combine their shares then they can recover a. In other words $OpenShare$ should return a if $|S| > \mathfrak{t}$ and all input shares are honestly entered, whilst $OpenShare$ should return $\perp$ if $|S| \leq \mathfrak{t}$. What happens when $|S| > \mathfrak{t}$ and the input set contains invalid share values depends on the precise secret sharing scheme being used.

Over finite fields the "classic" threshold secret sharing scheme is that of Shamir [Sha79]. In this section we define a Shamir–like Secret Sharing scheme over the Galois Ring $GR(q, F)$ for $\mathfrak{n}$ players with threshold $\mathfrak{t}$. We will do this by making the shares be the coordinates of a codeword in the Reed–Solomon code $RS_{n, \mathfrak{t}+1}$.

See the algorithms for sharing and reconstruction given in Figure 34. These algorithms will be used as sub–components of other fully robust algorithms later. Notice, that the opening algorithm, given in Figure 34, *may* abort if any of the $\mathfrak{n}$-parties send in a share value which is inconsistent; this is gauranteed to happen if $\mathfrak{t} < \mathfrak{n}/2$. In addition $OpenShare$ will abort if the shared value is not in $\mathbb{Z}/(q)$, but is in $GR(q, F) \setminus \mathbb{Z}/(q)$.

The construction in Figure 34 is justified as follows: We first take an exceptional sequence $\mathcal{E}$ of size $\mathfrak{n}$ not containing any element equal to zero (or equal to zero mod p when $q = p^k$). For example one can

---

Syndrome Decoding over $GR(q, F)$

$SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$:
1. If $q = p^k$ then
   (a) $\mathbf{e} \leftarrow \mathbf{0}$.
   (b) $S_{\mathbf{e}}^{(0)}(Z) \leftarrow S_{\mathbf{e}}(Z)$.
   (c) For $j \in [0, \ldots, k-1]$ do
       i. $s(Z) \leftarrow S_{\mathbf{e}}^{(j)}(Z)/p^j \pmod{p}$.
       ii. $\mathbf{e}^{(j)} \leftarrow SynDecode^{\mathbb{F}}(p, s(Z))$. Treat the resulting vector $\mathbf{e}$ as an error vector in $GR(q, F)$.
       iii. $\mathbf{e} \leftarrow \mathbf{e} + p^j \cdot \mathbf{e}^{(j)}$.
       iv. Update $S_{\mathbf{e}}(Z)$ via

$$S_{\mathbf{e}}^{(j+1)}(Z) \leftarrow S_{\mathbf{e}}^{(j)}(Z) - p^j \cdot \left(\sum_{i=1}^{n} \frac{e_j^{(i)}}{L_i(\alpha_i) \cdot (1 - \alpha_i \cdot Z)} \pmod{Z^r} \right).$$

2. Else
   (a) For $i \in [1, \ldots, k]$ do
       i. $\mathbf{e}_i \leftarrow SynDecode^{GR}(p_i, S_{\mathbf{e}}(Z) \pmod{p_i})$.
   (b) $\mathbf{e} \leftarrow CRT(\mathbf{e}_1, \ldots, \mathbf{e}_t)$.
3. Return $\mathbf{e}$.

$Correct^{GR}(q, \mathbf{r})$:
1. Compute the syndrome polynomial $S_{\mathbf{r}}(Z) = S_{\mathbf{e}}(Z)$ via equation (27).
2. $\mathbf{e} \leftarrow SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$.
3. $\mathbf{c} \leftarrow \mathbf{r} - \mathbf{e}$.
4. Return $\mathbf{c}$.

Figure 33: Decoding via Syndromes for Reed–Solomon Codes for the Galois Ring $GR(q, F)$..

The Secret Sharing Scheme $\langle x \rangle$

$Share(a)$: Given $a \in \mathbb{Z}/(q)$ this produces a sharing, i.e. values $\langle a \rangle_i \in R = GR(q, F)$
1. Generate a random polynomial $g_a(X) \in R[X]$ of degree at most $\mathfrak{t}$ such that $g_a(0) = a$.
2. Define $\langle a \rangle_i = g_a(\alpha_i)$.

$OpenShare(\{\langle a \rangle_i\}_{i \in S})$:
1. If $|S| < \mathfrak{t}$ the *abort*.
2. Compute the polynomial, where $\delta_i^S(Z)$ is from equation (25),

$$g_a(X) \leftarrow \sum_i \langle a \rangle_i \cdot \delta_i^S(Z).$$

3. If $\deg g_a(Z) > \mathfrak{t}$ then *abort*.
4. If $g_a(0) \notin \mathbb{Z}/(q)$ then *abort*.
5. Return $g_a(0)$.

Figure 34: The Secret Sharing Scheme $\langle x \rangle$.

select $\mathcal{E}$ as a subset of the canonical maximal exception sequence minus zero, i.e. $\mathcal{E} \subseteq GR(p, F) \setminus \{0\}$. Since we have already imposed the constraint $\mathfrak{n} > p^{\mathfrak{d}}$, by Parameter Choice 3, such an exceptional sequence exists.

To secret share an element $s \in GR(q, F)$, the dealer selects t elements $a_i \in GR(q, F)$ and forms the

polynomial
$$G(Z) = s + a_1 \cdot Z + a_2 \cdot Z^2 + \ldots a_t \cdot Z^t.$$

The share values are the evaluations of the polynomial $G(Z)$ at the points $\{\alpha_1, \ldots, \alpha_\mathfrak{n}\} = \mathcal{E}$ in the exceptional sequence, i.e.
$$s_i = G(\alpha_i).$$

The value s_i is given to player $\mathcal{P}_i$. We see that the vector of share values **s** is a codeword in $RS_{\mathfrak{n}, t+1}$.

The reconstruction works, when the input share values into *OpenShare* are valid, follows from La-grange interpolation. In addition, any subset $J \subseteq \{1, \ldots, \mathfrak{n}\}$ of size less than or equal to t can determine no information about the secret s from their shares.

Sometimes we may share with respect to a degree different d from t, in which case we write $\langle s \rangle^d$. Thus $\langle s \rangle$ is a shorthand for $\langle s \rangle^t$. Although now one needs $d + 1$ valid shares in order to reconstruct.

The secret sharing is linear over $GR(q, F)$, i.e. given sharings $\langle x \rangle$ and $\langle y \rangle$ and constants $\alpha, \beta, \gamma \in GR(q, F)$ one can has, for all i,
$$\langle \alpha \cdot x + \beta \cdot y + \gamma \rangle_i = \alpha \cdot \langle x \rangle_i + \beta \cdot \langle y \rangle_i + \gamma.$$

Hence, linear operations can be computed locally for "free".

If we let b denote the number of adversaries in our set $\{\mathcal{P}_1, \ldots, \mathcal{P}_\mathfrak{n}\}$ then this means that, for security, we must have
$$b \leq t,$$

i.e. we can tolerate at most t bad guys in order to preserve privacy. The remaining issue is what happens when bad players lie about their shares.

7.1.3.1 Error Detection: The bad–actors can of course pass on invalid shares to someone who aims to reconstruct the secret. Thus the received share vector **s**, obtained by someone wishing to recover the underlying secret, may contain errors. Thus the received vector **s** may not lie in $RS_{\mathfrak{n}, t+1}$ at all; it may be perturbed by an error vector. In Section 7.1.2 we saw that this can be **detected** if the computed syndrome polynomial $S_\mathbf{s}(Z)$ is equal to zero or not.

For the sharing $\langle s \rangle$, the maximum number of errors we can detect is bounded by $\mathfrak{n} - \mathfrak{v} = \mathfrak{n} - (t + 1)$, thus this is also an upper bound on the number of adversaries. So if we are only interested in detecting whether an adversary has sent an invalid share we have the upper bound

$$b \leq \mathfrak{n} - (t + 1).$$

Combining this with the equation $b \leq t$, to ensure secrecy, we find

$$2 \cdot b \leq b + t \leq \mathfrak{n} - 1,$$

or $b < \mathfrak{n}/2$, i.e. we can tolerate at most $\mathfrak{n}/2$ as adversaries if we wish to only detect errors in our secret sharing. This is effectively what happens in algorithm *OpenShare* in Figure 34; if $t < \mathfrak{n}/2$ we can detect errors introduced by the adversary into the *OpenShare* algorithm.

When using the sharings $\langle s \rangle^d$ of degree d, we obtain the equations $b \leq \mathfrak{n} - (d + 1)$ and $b \leq t$. In one situation which follows we will have $d = 2 \cdot t$ and so

$$3 \cdot b \leq b + 2 \cdot t = b + d < \mathfrak{n},$$

or $b < \mathfrak{n}/3$, i.e. we can tolerate at most $\mathfrak{n}/3$ as adversaries if we wish to only detect errors in our secret sharing of degree $2 \cdot t$.

7.1.3.2 Error Correction: If we are interested in recovering the share, even in the presence of errors, we need to apply the error correction methods of Section 7.1.2. This require that the number of errors is bound by $\lfloor(\mathfrak{n} - \mathfrak{v})/2\rfloor$, for our $\langle s \rangle$ sharings. In other words we have

$$2 \cdot b \leq \mathfrak{n} - (\mathfrak{t} + 1).$$

Again, combining this with the equation $b \leq \mathfrak{t}$ we find

$$3 \cdot b \leq \mathfrak{t} + 2 \cdot b \leq \mathfrak{n} - 1,$$

or $b < \mathfrak{n}/3$, i.e. the number of bad guys needs to be bounded by $\mathfrak{n}/3$.

When using the sharings $\langle s \rangle^d$ of degree d we obtain the equations $2 \cdot b \leq \mathfrak{n} - (d + 1)$ and $b \leq \mathfrak{t}$. In the important case of $d = 2 \cdot \mathfrak{t}$ this becomes

$$4 \cdot b \leq 2 \cdot b + 2 \cdot \mathfrak{t} = 2 \cdot b + d < \mathfrak{n},$$

i.e. $b < \mathfrak{n}/4$. This is why we need to introduce the threshold profiles in the set $nLarge$, i.e. $nMfH^+, nLfH^+$ and $nEfH^+$, in Section 6.1.

7.1.3.3 Randomness Extraction: A common trick used in many MPC protocols, see for example [DN07], is to extract c random sharings from an input set of r possibly random sharings, where $r > c$. This technique was shown to extend to the Galois Ring situation in [ACD$^+$19]. The idea being is that if at least c of the input sharings are truly random, then *all* of the output sharings are random. In particular if the input sharings are produced by $r = \mathfrak{n}$ players independently, or which we know at most $\mathfrak{t}$ are potentially dishonest, then we can output $n - \mathfrak{t}$ sharings which are random. This assumes the input sharings are valid, i.e. valid degree $\mathfrak{t}$ sharings, and the only thing the adversary can do is know the values of the shared values for the $\mathfrak{t}$ sharings that it input into the procedure.

This "randomness extraction" is performed by using the Vandermonde matrix $M_{r,c}$ from earlier, where the entries are generated from an exceptional sequence of size $r = \mathfrak{n}$. Then given r input sharings $\langle s_1 \rangle, \ldots, \langle s_r \rangle$ we can produce, using only local operations, the c output sharings in our randomness extraction procedure by computing

$$\begin{pmatrix} \langle t_1 \rangle \\ \ldots \\ \langle t_c \rangle \end{pmatrix} = M_{r,c}^\top \cdot \begin{pmatrix} \langle s_1 \rangle \\ \vdots \\ \langle s_r \rangle \end{pmatrix}.$$

Applying the super–invertibility property, with $r = \mathfrak{n}$ and $c = \mathfrak{n} - \mathfrak{t}$, and using the sub-matrix of c rows and columns where we range over the (unknown) row set consisting of the rows corresponding to shares which come from honest parties, we see that the output distribution is of random values. In particular the $\mathfrak{t}$ adversarial party has no control, or information, over the random values in the output sharings.

7.1.4 Commitment Schemes

We will need a commitment scheme in various places. The one we choose will be secure in the Random Oracle Model (ROM), and thus uses a hash function. The hash function

$$Hash^{2 \cdot sec} : \{0, 1\}^* \longrightarrow \{0, 1\}^{2 \cdot sec}$$

has hash output length $(2 \cdot sec)$–bits. As we target $sec = 128$ in this document, SHA2–256 or SHA3–256 will be sufficient[16]. The commitment scheme is defined in Figure 35; note, the addition of the pre–fix $DSep(COMM)$ for domain separation. If we want to specify the randomness externally to the commitment then we write $Commit(m, i, sid, rid; o)$.

Commitment Scheme

$Commit(m, i, sid, rid)$: On input of a message $m \in \{0, 1\}^*$, party ID i, session ID sid and round ID rid this proceeds as follows, to produce the commitment c and the opening information o.
  1. Generate a random bit string as opening information $o \leftarrow \{0, 1\}^{sec}$.
  2. Compute the commitment $c \leftarrow Hash^{2 \cdot sec}(DSep(COMM)\|sid\|rid\|m\|o)$.
  3. Return (c, o).

$Verify(c, o, m, i, sid, rid)$: This verifies correctness of the commitment.
  1. Compute $c' \leftarrow Hash^{2 \cdot sec}(DSep(COMM)\|i\|sid\|rid\|m\|o)$.
  2. If $c' \neq c$ then return $false$.
  3. Else return $true$.

Figure 35: Commitment Scheme.

7.1.5 Bit Generation

All our MPC protocols are required to generate random bits. This is done via classical/folk–lore methods for which we now describe the underlying mathematics (the precise implementation of this in our MPC protocols will be delayed till later).

The goal is to produce a bit modulo q using only additions and multiplications (to some extent). In the prior literature, for example [DKL+13] and [OSV20], methods are given to generate bits in $\mathbb{Z}/(q)$ via an MPC protocol; the first reference being when q is an odd prime, and the second reference being for when q is a power of two. When applying these algorithms we will be working with the Galois Ring $GR(q, F)$. The extension of these two prior methods to the Galois Ring context, and the optimizations we present below from the original presentation are immediate. Note, in this section we are not thinking of the case when $q = p_1 \cdots p_k$, as this will require a special treatment later; thus we only think of $q = 2^k$ and q a prime.

7.1.5.1 $Solve(v)$: Before progressing we discuss solving quadratic equations in the Galois Ring, when q is a power of two, i.e. $q = 2^k$, see Figure 36. For our Galois Rings we define the trace and half–trace functions by

$$Tr(x) = \sum_{j=0}^{(\mathfrak{d}-1)} x^{2^j},$$

$$\overline{Tr}(x) = \sum_{j=0}^{(\mathfrak{d}-1)/2} x^{2^{2j}}.$$

Our method to generate bits, when q is a power of two, requires access to a function $Solve(v)$, for $v \in GR(q, F)$, which solves the equation

$$X^2 + X = v \pmod{q} \tag{28}$$

---

[16] In our implementation we use SHA3–256.

where $q = 2^k$, assuming such a solution exists. Such a solution exists if we have $Tr(v) = 0$ (mod 2).

To solve this equations requires two steps: First we solve the equation modulo 2, i.e. in the finite field $\mathbb{F}_{2^{\mathfrak{d}}}$, and then we lift this solution using Hensel's Lemma to the whole ring $GR(q, F)$. To solve

$$X^2 + X = v \quad \text{(mod 2)}, \tag{29}$$

i.e. in $\mathbb{F}_{2^{\mathfrak{d}}}$, we apply the classical method, to be found in [BSS99, page 26], which we denote by the function $Solve_1(v)$, which depends on the parity of $\mathfrak{d}$.

$\mathfrak{d}$ Odd: In this case we compute

$$x_0 = Solve_1(v) = \overline{Tr}(v) \quad \text{(mod 2)}.$$

$\mathfrak{d}$ Even: This case is slightly more complex. We first find an element $\delta \in \mathbb{F}_{2^{\mathfrak{d}}}$ such that $Tr(\delta) = 1$. This is in fact easy, as half the elements in $\mathbb{F}_{2^{\mathfrak{d}}}$ have trace one. We then can write

$$x_0 = Solve_1(v) = \sum_{i=0}^{\mathfrak{d}-2} \Big(\sum_{j=i+1}^{\mathfrak{d}-1} \delta^{2^j} \Big) \cdot v^{2^i} \quad \text{(mod 2)}.$$

The second step is to solve lift the solution to equation (29), to a solution of equation (28) by executing the following recursion $\lceil \log_2 k \rceil$ times

$$x_0 = Solve_1(v),$$

$$x_{n+1} = \frac{x_n^2 + v}{1 + 2 \cdot x_n} \quad \text{(mod } q\text{)}, \quad \text{for } n \geq 0.$$

This appears to require a full outer Newton iteration in order to find the successive x_i, and a full inner Newton iteration to find the inverse of $(1 + 2 \cdot x_n)$. However, this initial $O(\lceil \log_2 k \rceil^2)$ estimate of operations can be replaced with $O(\lceil \log_2 k \rceil)$ iterations, using the algorithm in Figure 36. The algorithms correctness follows from standard Hensel lifting/Newton iteration arguments.

$Solve(v)$

1. $x \leftarrow Solve_1(v)$.
2. $y \leftarrow 1$.
3. For $i \in [1, \ldots, \lceil \log_2 k \rceil]$ do
    (a) $m \leftarrow 2^{2^i}$.
    (b) $z \leftarrow 1 + 2 \cdot x$ (mod m).
    (c) $y \leftarrow y \cdot (2 - z \cdot y)$ (mod m).
    (d) $y \leftarrow y \cdot (2 - z \cdot y)$ (mod m).
    (e) $x \leftarrow (x \cdot x + v) \cdot y$ (mod m).
4. Return x (mod q).

Figure 36: Solving $X^2 + X = v$ (mod q) Using Hensel Lifting.

7.1.5.2 $Sqrt(v, p)$: We also need to provide a method to produce a square root of an element $v \in GR(p, F)$ when p is an odd prime. There are standard methods to produce square roots in the finite field $\mathbb{F}_{p^{\mathfrak{d}}}$, see for example [Coh93, Sho05]. We denote these methods by $Sqrt(v, p)$. The classic algorithms for this are given in Figure 37; the algorithm when $p^{\mathfrak{d}} \equiv 1$ (mod 4) is called the Tonelli–Shanks algorithm.

Sqrt

Sqrt(v, p): Assumes p is odd prime, and $v \in GR(p, F)$ is indeed a square.
1. If $v = 0$ then return v.
2. If $p^{\mathfrak{d}} \equiv 3 \pmod 4$ then
   (a) $s \leftarrow v^{(p^{\mathfrak{d}}+1)/4}$.
   (b) Return s.
3. Else
   (a) Write $p^{\mathfrak{d}} - 1 = 2^S \cdot Q$ with Q odd, and find an $z \in GR(p, F)$ which is a quadratic non-residue.
       This step can obviously be done as pre-processing when one sets up the system.
   (b) $M \leftarrow S, c \leftarrow z^Q, t \leftarrow v^Q, r \leftarrow v^{(Q+1)/2}$.
   (c) While $t \neq 1$ do
       i. Find smallest i such that $t^{2^i} = 1$ with $0 < i < M$.
       ii. $b \leftarrow c^{2^{M-i-1}}$.
       iii. $c \leftarrow b^2$.
       iv. $t \leftarrow t \cdot b^2$
       v. $r \leftarrow r \cdot b$.
       vi. $M \leftarrow i$.
   (d) Return r.

Figure 37: Finite Field Square Root Algorithm.

7.1.5.3 Random Bits p Odd Prime: This is a classic method, see Figure 38, which relies on the fact that the squaring map is a $2 : 1$ mapping when p is a prime. The method seems first to have been applied in the MPC context in [DKL$^+$13], in the case of prime p and the degree one Galois Ring. Extending this to the general case of a general finite field extension $\mathbb{F}_{p^{\mathfrak{d}}} = GR(p, F)$ is immediate, and is given in Figure 38. Note this method is never implemented, it is just described here as a guide to explain the protocol later.

Bit Generation: p Odd Prime

1. $a \leftarrow GR(p, F)$, such that $a \neq 0 \pmod p$.
2. $s \leftarrow a^2 \pmod p$.
3. $c \leftarrow$ *Sqrt*(s, p).
4. $v \leftarrow a/c \pmod p$. This makes v equal to -1 or 1 modulo q.
5. $b \leftarrow (1 + v)./2 \pmod p$. The value of b will be 0 or 1 modulo p.
6. Return b.

Figure 38: Random Bit Generation when p is an Odd Prime.

Generalizing this to $q = p_1 \ldots p_k$ is possible by the Chinese Remainder Theorem. The obvious trick is to first select $a \leftarrow GR(q, F)$ to be not equal to zero modulo any prime divisor of q. Then one obtains c by applying the Chinese Remainder Theorem to the output of the t square roots modulo each prime divisor of q, i.e. we compute *Sqrt*(s, q) The problem then is that the final bit b will definitely be equal to zero or one modulo each prime p_i, but it will not necessarily equal the same bit modulo each prime divisor, i.e. we have

$$b \pmod{p_i} \in \{0, 1\} \text{ for all } i$$

but not

$$b \pmod{p_i} = b \pmod{p_j} \text{ for } i \neq j.$$

Thus this naive generalization does not work. Later we will find a solution to this problem in our MPC protocol for threshold profiles *nSmall*.

119

7.1.5.4   Random Bits q Power of Two: When q is even we cannot use the above technique, since the squaring map is now a $4:1$ mapping. Instead we utilize a technique from [OSV20], which we modify a little. It is functionally the same but written down in a slightly different manner, which is easier for our application. This simplification was first presented in [DDE$^+$23]. What is nice, from an MPC perspective, about the even q case is that we do not need to loop to produce a non–zero value. This then gives us the algorithm given in Figure 39. Again this algorithm is never implemented explicitly, it is here to act as a guide to the MPC implementation later.

Bit Generation: q Even

1. $a \leftarrow \mathcal{G}$.
2. $v \leftarrow a + a^2 \pmod{q}$.
3. $r \leftarrow Solve(v)$.
4. $d \leftarrow (-1 - 2 \cdot r) \pmod{q}$.
5. $b \leftarrow (a - r)/d \pmod{q}$.
6. Return b.

Figure 39: Random Bit Generation when q is Even.

Example: We go through a small worked example to demonstrate this actually works using the simple ring $\mathbb{Z}/(q)$, for $q = 2^3 = 8$.
1. $a \leftarrow \mathbb{Z}/(2^3)$. So take, for example $a = 3$.
2. $v \leftarrow a + a^2 \pmod 8$. So in our example $v = 4$.
3. Applying $r \leftarrow Solve(v)$ gives us $r = 3$.
4. $d \leftarrow -1 - 2 \cdot r$, gives us $d = 1$.
5. $b \leftarrow (a - r) / d \pmod 8 = (3 - 3) / 1 \pmod 8 = 0$.

7.1.6   TreePRG

A *TreePRG* this is a Key Encapsulation Mechanism (KEM) which assigns one key to each leaf of a binary tree of depth d; where all of the leaves are derived from a single master seed *seed*. However, the creator of the *TreePRG* can produce a punctured seed D, which is the output of *TreePRG.Punc*(*seed*, d, T), for some subset $T \subset \{1, \ldots, 2^d\}$. The punctured seed allows the holder to evaluate all leaf nodes except those in T. The key advantage is that the size of the punctured seed is bounded by

$$|D| \leq |T| \cdot \log_2 \left(\frac{2^d}{|T|} \right) \cdot sec. \tag{30}$$

Intuitively, $\log_2 \left(\frac{2^d}{|T|} \right)$ is the depth of the largest possible sub–tree that can be expanded using a seed and there are no more than $|T|$ of these. By ignoring leaves we can produce *TreePRG*'s which encode a non–power of two number of leaves in an obvious manner.

  Our description of the method, see Figure 40, for constructing a TreePRG is adapted from a version given in the full version of [GGHAK22], where a method for sets T of size one is given. The extension to sets T of arbitrary size is derived from the presentation in [NNL01]. The description uses of a pseudo–random generator $PRG : \{0,1\}^{sec} \longrightarrow \{0,1\}^{2 \cdot sec}$. This is constructed as follows

$$PRG(seed) := \begin{cases} XOF \leftarrow XOF.Init(seed, DSep(TPRG)) \\ y \leftarrow XOF.Next(2 \cdot sec) \\ \text{Return } y \end{cases} \tag{31}$$

The notation in our algorithm description makes use of labels assigned to each node of the tree. To a

binary tree of depth d we assign labels to each node as follows:

- The root node has an empty label $\emptyset$.
- For each internal node with label lab the left child node is given label $lab\|0$ and the right child node is given label $lab\|1$.
- The leaf nodes have labels of binary length d and we can associate each leaf label with an integer in the range $[0, \ldots, 2^{d-1}]$ by treating the label as the binary representation of the given integer. With the integers matching the leafs in consecutive order.

TreePRG

TreePRG.Gen($seed, d$):
   On input of $seed \in \{0, 1\}^{sec}$ and a depth d this constructs a binary tree of depth d. The output is the set of seeds and labels of the leaves.
   1. Return *TreePRG.GenSub*($seed, \emptyset, d, 0$).

TreePRG.GenSub($seed, lab, d, i$):
   1. If $i = d$ then return ($seed, lab$).
   2. ($seed_0, seed_1$) $\leftarrow PRG(seed)$.
   3. $a \leftarrow$ *TreePRG.GenSub*($seed_0, lab\|0, d, i+1$).
   4. $b \leftarrow$ *TreePRG.GenSub*($seed_1, lab\|1, d, i+1$).
   5. Return (a, b).

TreePRG.PuncSub(D, lab, T, i):
   1. If lab not the prefix of any integer in T then append the seed $seed_{lab}$ to D and return D.
   2. If $i = d$ then return D.
   3. $D \leftarrow$ *TreePRG.PuncSub*($D, lab\|0, T, i+1$).
   4. $D \leftarrow$ *TreePRG.PuncSub*($D, lab\|1, T, i+1$).
   5. Return D.

TreePRG.Punc($seed, d, T$):
   On input of $seed \in \{0, 1\}^{sec}$, a depth d, and a set $T \subset \{0, \ldots, 2^d - 1\}$, this constructs a punctured ordered list D of seeds as follows:
   1. $D \leftarrow [\,]$.
   2. $D \leftarrow$ *TreePRG.PuncSub*($D, \emptyset, T, 0$)

TreePRG.GenPunc(D, T, d):
   Given a set of punctured seeds D, which have been output by *TreePRG.Punc*, for a tree of size d and set T, this outputs the seeds corresponding to all leaves which are not in the set T.
   1. Set $S \leftarrow \{\,\}$.
   2. For each $seed_{lab}$ in the ordered list D:
       (a) Obtain the corresponding lab and depth i from running the algorithm *TreePRG.Punc*($seed, d, T$) "in ones head".
       (b) $S \leftarrow S \cup$ *TreePRG.GenSub*($seed_{lab}, lab, d, d - i$).
   3. Return S.

Figure 40: Tree-based Pseudo-random Generator.

7.2 Layer One

We summarize the relationship between the protocols in Layer One in Figure 41. In the diagrams in this section orange refers to a component which is not actively secure, a cyan refers to a component which is actively secure with abort, and a green refers to a component which is robustly secure. A component which is not a protocol, and which lives on Layer Zero, is denoted with white.
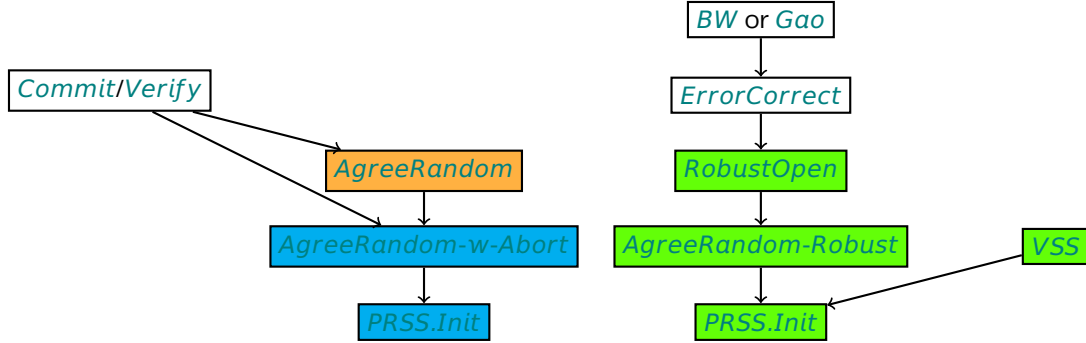
Figure 41: Different versions of *PRSS.Init* in the Threshold Profile Category *nSmall*. These protocols assume **synchronous** networks. .

7.2.1 Broadcast

As discussed in Section 4, at various points we require a sub–protocol which provides a reliable broadcast over a synchronous network. Bracha's original protocol for broadcast works in asynchronous networks, we however will only apply broadcast in synchronous networks; thus we can make some small sim–plifications to Bracha's original protocol [Bra87]. The modifications to Bracha's protocol are given in Figure 42, where we take the modifications from https://hackmd.io/@alxiong/bracha-broadcast.

To recap, from Section 4.5, the protocol in Figure 42 has the following properties, when $\mathfrak{t} < \mathfrak{n}/3$,

- Every honest party will terminate will either a message *m*, or a value ⊥.
- Any two honest parties that terminate output the same value, and if *S* is honest then that message is *m*.

The use of $\Omega(\mathfrak{t})$ rounds in this case is optimal in our case of non–randomized synchronous broadcast protocols without PKI, see for example [DS83, FL82].

The protocol should be interpreted in the following way: Each round is processed within a time limit Δ, and if no messages are received then the party moves onto the next round. Thus the protocol will output ⊥ if the sender not only sends inconsistent values, but also sends nothing at all. To ensure parties are in synchronization (and to avoid large time outs Δ) if in a round a player is asked to send no message, then it simply sends "no message" to all other players. Thus honest parties know when another honest party is not sending a message in this round.

If a party exceeds the time limit in round *k* without receiving data from a specific party $\mathcal{P}_i$, then the party should assume that $\mathcal{P}_i$ is adversarial and it should not be waited for in rounds following on from *k*. To avoid sending too many large messages in the *vote* phase of Figure 42 we only send, and verify, hashes of messages. These hashes are computed via the hash function $Hash^{2 \cdot sec}$.

In practice we will usually call *Synch-Broadcast* in parallel, i.e. every player in the set $\{\mathcal{P}_1, \dots, \mathcal{P}_{\mathfrak{n}}\}$ will simultaneously be trying to broadcast some data. The implementation thus allows passing a set of senders $\mathcal{S} \subseteq \mathcal{P}$ while merging communication rounds across the parallel executions when possible.

The execution time of the protocol is upper bounded by $\Delta \cdot (3 + \mathfrak{t} + 1)$, as we require executing $3 + \mathfrak{t}$ rounds of communication, thus efficient implementation and a suitable choice of Δ (which is probably network/ping–time related) is important.

Synchronous Reliable Broadcast

$Synch\text{-}Broadcast(S, m)$: The protocol executes in the following rounds, where r is the round number

1. Party S sends $(send, m)$ to $\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}$.
2. Party $\mathcal{P}_i$ if they receive $(send, m)$ from S then they send $(echo, m)$ to $\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}$.
3. If $\mathcal{P}_i$ receives $(echo, m)$ from at least $\mathfrak{n} - \mathfrak{t}$ parties, then send $(vote, Hash^{2 \cdot sec}(DSep(BRACH)\|m))$ to $\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}$.

4 to $3 + \mathfrak{t}$. If $\mathcal{P}_i$ has received up to this round a $(vote, Hash^{2 \cdot sec}(DSep(BRACH)\|m))$ from $\mathfrak{t} + r - 3$ distinct parties (where r is the round number), and it has not voted before then, send $(vote, Hash^{2 \cdot sec}(DSep(BRACH)\|m))$ to $\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}$.

$3 + \mathfrak{t} + 1$. If $\mathcal{P}_i$ has received the value $(vote, Hash^{2 \cdot sec}(DSep(BRACH)\|m))$ from at least $\mathfrak{n} - \mathfrak{t}$ distinct players then output m, otherwise output $\perp$.

Figure 42: Variant of Bracha's Reliable Broadcast Protocol for Synchronous Networks..

The reader should note that even if one replaces the above synchronous version of Bracha with an asynchronous version we do not obtain fully asynchronous MPC protocols. We adopt a synchronous version of Bracha, as all our offline MPC protocols need to work over synchronous networks. We only use a broadcast primitive in the Offline phase of our MPC protocols; thus use of a synchronous version of Bracha's protocol does not affect the asynchronous nature of online phase of our MPC protocol.

7.2.2 Dispute Control

Our robust MPC protocols work in the Dispute Control framework, as given in [DNO7]. At all points each party maintains two sets *Dispute* and *Corrupt*. The set *Dispute* consists of pairs of players $\{\mathcal{P}_i, \mathcal{P}_j\}$, with $i < j$, with the following properties:

- All players agree on the value of *Dispute*.
- If $\{\mathcal{P}_i, \mathcal{P}_j\} \in Dispute$ then either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupt.

We let $Dispute_i$ be defined by

$$Dispute_i = \big\{ \{\mathcal{P}_j, \mathcal{P}_k\} \in Dispute : \mathcal{P}_j = \mathcal{P}_i \text{ or } \mathcal{P}_k = \mathcal{P}_i \big\}$$

i.e. it is the set of players with which $\mathcal{P}_i$ is in dispute with. Hence, $Dispute = \cup_i Dispute_i$. We also set $Agree_i = \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\} \setminus Dispute_i$. We require the following properties of our sub–protocols:

1. A protocol can never halt due to a dispute between $\mathcal{P}_i$ and $\mathcal{P}_j$ if $\{\mathcal{P}_i, \mathcal{P}_j\}$ is already in *Dispute*.
2. If a protocol does not generate a dispute then it terminates with the correct result.
3. If it does generate a dispute then one can re–run the protocol with the larger set of disputes.

We also make use of a set called *Corrupt*. The set *Corrupt* details which parties the honest parties have agreed upon to be dishonest. It has the following properties:

- All honest players agree on the value of *Corrupt*.
- If $\mathcal{P}_i \in Corrupt$ then $\mathcal{P}_i$ really is corrupt.

An important property, which we will implicitly assume from now on, is that if ever the set $Dispute_i$ becomes so large that $|Dispute_i| > \mathfrak{t}$, then all players add $\mathcal{P}_i$ into the set *Corrupt*. In addition if $\mathcal{P}_i \in Corrupt$ then $\{\mathcal{P}_i, \mathcal{P}_j\} \in Dispute$ for all j.

We can modify the synchronous $Synch\text{-}Broadcast(S, m)$ operation from Figure 42 to update the set *Corrupt*. This is done in Figure 43. The logic behind this modification is that if an honest party receives $\perp$ from an execution of $Synch\text{-}Broadcast(S, m)$, then it knows the sender is dishonest, and since the broadcast is reliable we know that all honest parties will agree on the fact that the sender is dishonest.

Synch-Broadcast($S, m, Corrupt$):
  1. Party S executes *Synch-Broadcast*(S, m).
  2. Party $\mathcal{P}_i$ will receive a message m'.
  3. If $\mathcal{P}_i$ receives $m' = \perp$ then S is added to the set *Corrupt*.
  4. Party $\mathcal{P}_i$ accepts m' as the value sent.

Figure 43: Variant of Synchronous Reliable Broadcast which updates the set *Corrupt*.

7.2.3 Robust Opening

As well as the opening procedure *OpenShare* from Figure 34 we also define an opening procedure called *RobustOpen* in Figure 44, which will robustly open the shared value, depending on the relationship between $\mathfrak{t}$ and $\mathfrak{n}$, and the underlying network properties. We present the robust opening protocol for Shamir sharing of arbitrary degree d. When $d = \mathfrak{t}$ robust opening is only available when $\mathfrak{t} < \mathfrak{n}/3$. Note that, for asynchronous networks and $d = \mathfrak{t} < \mathfrak{n}/4$ we can execute potentially less computational steps than for the case $d = \mathfrak{t} < \mathfrak{n}/3$ by simply waiting for more data to arrive. The need for a general d is that at some points we need to robustly open a sharing of degree $d = 2 \cdot \mathfrak{t}$, despite there only being at most $\mathfrak{t}$ adversaries.

Note that, in the waiting steps in lines 3a and 4(a)i of protocol *RobustOpen*, we treat the implicit $\perp$ "received" from (locally) known malicious parties in line 2 as actual received values. Also note, that the call to *ErrorCorrect* in the algorithm can (for efficiency) also update the set of (locally) known corrupt parties to the reconstructing party[17].

7.2.3.1 Optimization via the King Paradigm: At various points in our protocol we need to robustly open a sharing $\langle x \rangle^d$ to all players, which we know is a valid degree $d \le 2 \cdot \mathfrak{t}$ sharing. In [DN07][Section 4.6], a method is given which batches $\ell = n - 2 \cdot \mathfrak{t} - 1$ such operations into one operation. The advantage of this is that it reduces bandwidth compared to our standard *RobustOpen* method above, the disadvantage is that it costs an additional round of communication. Thus the trade off depends heavily on the associated ping–time between the parties. In practice the increase in the number of rounds may not be so much of a gain given the relatively small value of $\mathfrak{n}$ we expect to run our MPC protocols with. The usual name for this optimization is to use the "king–paradigm", as messages are combined and sent to a "king" who then opens them. For those interested we outline the optimization here, but we will not use it.

The optimization utilizes the Vandermonde matrix $M_{\mathfrak{n},\ell}$ as an expander, instead of as a extractor. We think of the operation $\mathbf{y} \leftarrow \mathbf{x} \cdot M_{\mathfrak{n},\ell}^\top$ as expanding a message $\mathbf{x} \in R^\ell$ into a codeword $\mathbf{y} \in R^{\mathfrak{n}}$, where $R = GR(q, F)$ is our Galois Ring. By the choice of parameters we know that any vector with at most $\mathfrak{t}$ errors can be corrected. We (by abuse of notation) refer to the associated decoding algorithm (which can be done robustly) as $\mathbf{x} \leftarrow$ *ErrorCorrect*($q, \mathbf{y}$).

---

[17]We do not need consensus on this set of corrupt parties for future calls to *RobustOpen*.

RobustOpen

This protocol depends on the ratio between d, $\mathfrak{t}$, and $\mathfrak{n}$, and whether the underlying net-work is synchronous or asynchronous. It is run either by a player $\mathcal{P} = \mathcal{P}_i$ who already holds their share $\langle a \rangle_i^d$. (we call this Case A), or by an external player $\mathcal{P}$ (which we call Case B).

RobustOpen$(\mathcal{P}, \langle a \rangle^d)$: In this variant only player $\mathcal{P}$ will obtain the value a.
  1. Player $\mathcal{P}_i$ sends $\langle a \rangle_i^d$ **privately** to player $\mathcal{P}$.
  2. Player $\mathcal{P}$ assumes it received $\perp$ from known malicious parties.
  3. If $d + 3 \cdot \mathfrak{t} < \mathfrak{n}$ and the network is asynchronous, or $d + 2 \cdot \mathfrak{t} < \mathfrak{n}$ and the network is synchronous
     (a) $\mathcal{P}$ waits until they have received $d + 2 \cdot \mathfrak{t}$ (case A) or $d + 2 \cdot \mathfrak{t} + 1$ (case B) share values $\{\langle a \rangle_j^d\}_j$. Note, in the synchronous case if no value is sent then it is treated as $\perp$.
     (b) They apply the *ErrorCorrect*$(q, \mathbf{s}, \mathfrak{t})$ algorithm from Figure 31 to the $d + 2 \cdot \mathfrak{t} + 1$ shares $\mathbf{s}$ they hold, to obtain a Reed–Solomon codeword with $\mathfrak{v} = d + 1$, and hence to robustly compute $G(Z)$.
     (c) Compute $a \leftarrow G(0)$ and return a to player $\mathcal{P}$.
  4. If $d + 2 \cdot \mathfrak{t} < \mathfrak{n}$ and the network is asynchronous
     (a) For $r \in [0, \dots, \mathfrak{t}]$ do
         i. $\mathcal{P}$ waits until $d + \mathfrak{t} + r$ (case A) / $d + \mathfrak{t} + r + 1$ (case B) shares have been received.
         ii. Apply *ErrorCorrect*$(q, \mathbf{s}, r)$ from Figure 31 on the $d + \mathfrak{t} + r + 1$ shares $\mathbf{s}$ they hold, to obtain a Reed–Solomon codeword with $\mathfrak{v} = d + 1$, assuming there are r errors in these shares, to obtain $G(Z)$.
         iii. If $G(Z)$ is a degree d degree poly then, if there are at least $d + \mathfrak{t} + 1$ shares (out of the $d + \mathfrak{t} + r + 1$ shares) which lie on the polynomial, then this is the correct polynomial so output the constant term $G(0)$ to player $\mathcal{P}$ and exit the loop. Note, this step requires just scanning the $d + \mathfrak{t} + r + 1$ shares and counting how may lie on the polynomial.

RobustOpen$(\mathcal{S}, \langle a \rangle^d)$: This is a short hand for all players executing *RobustOpen*$(\mathcal{P}_i, \langle a \rangle^d)$, for all $\mathcal{P}_i \in \mathcal{S} \subseteq \{\mathcal{P}_1, \dots \mathcal{P}_\mathfrak{n}\}$, in parallel. Hence, all players in $\mathcal{S}$ will obtain the value a. In this case, the communication of the shares in line 1 only needs to be done privately if $\mathcal{S} \neq \{\mathcal{P}_1, \dots, \mathcal{P}_\mathfrak{n}\}$.

Figure 44: Robust Opening Protocol when $d + 2 \cdot \mathfrak{t} < \mathfrak{n}$.

BatchRobustOpen

BatchRobustOpen$((\langle x_i \rangle^d)_{i=1}^\ell)$: This method assumes $d \leq 2 \cdot \mathfrak{t}$ and $\ell = \mathfrak{n} - 2 \cdot \mathfrak{t} - 1$.
  1. Locally compute
$$(\langle y_i \rangle^d)_{i=1}^\mathfrak{n} \leftarrow (\langle x_i \rangle^d)_{i=1}^\ell \cdot M_{\mathfrak{n}, \ell}^\top.$$
  2. For $i \in [1, \dots, \mathfrak{n}]$ in parallel execute
     (a) For all j, player $\mathcal{P}_j$ sends $\langle y_i \rangle_j^d$ to player $\mathcal{P}_i$.
     (b) Player $\mathcal{P}_i$ executes $y_i \leftarrow$ *ErrorCorrect*$(q, (\langle y_i \rangle_j^d)_{j=1}^\mathfrak{n})$.
     (c) Player $\mathcal{P}_i$ sends y_i to all players.
  3. All parties compute $\mathbf{x} \leftarrow$ *ErrorCorrect*$(q, \mathbf{y})$.

Figure 45: Robust Batch Opening.

7.2.4 Verifiable Secret Sharing

For the threshold profiles in category *nLarge* we need a standard Verifiable Secret Sharing (VSS) protocol in order to execute a *CoinFlip* protocol. In addition for the threshold profiles *nSmall* we need a standard VSS algorithm to agree on a small number of random values for use in a robust protocol to set up our

PRSS functions (see below). For threshold category *nSmall* if we can accept an initialization routine *PRSS.Init*() which is only active–with–abort secure, then the VSS is not needed.

We utilize in this work an adaption of the the four round VSS protocol from [GIKR01] for finite fields, to the Galois Ring setting. Our presentation of the protocol follows the description from [CCP22] (where it is labeled as Protocol 4GIKR–VSS–Sh); see Figure 46.

VSS

$VSS(\mathcal{P}_k, s, \mathfrak{t}, Corrupt)$. This is called by player $\mathcal{P}_k$ to verifiably share a secret $s \in R = GR(q, F)$ amongst the other players of degree $\mathfrak{t}$.

Round 1
1. Player $\mathcal{P}_k$ selects a bivariate polynomial $F(X, Y) \in R[X, Y]$ of degree $\mathfrak{t}$ in both variables such that $F(0, 0) = s$.
2. Define $F_i(X) := F(X, \alpha_i)$ and $G_i(Y) := F(\alpha_i, Y)$, for all i.
3. Send $F_i(X)$ and $G_i(X)$ **privately** to player $\mathcal{P}_i$.
   Note, if $\mathcal{P}_i$ receives nothing within the round time–out time of Δ then it interprets the result as $F_i(X) = G_i(X) = 0$.
4. Each $\mathcal{P}_i$ picks a random value $r_{i,j} \in R$, one for each player $\mathcal{P}_j$, and it sends $r_{i,j}$ **privately** to player $\mathcal{P}_j$.
   Again, if $\mathcal{P}_j$ receives nothing within the round time–out time of Δ from player $\mathcal{P}_i$, then it interprets the result as $r_{i,j} = 0$.

Round 2
5. Player $\mathcal{P}_i$ computes $a_{i,j} \leftarrow F_i(\alpha_j) + r_{i,j}$ and $b_{i,j} \leftarrow G_i(\alpha_j) + r_{j,i}$.
6. Player $\mathcal{P}_i$ executes $Synch\text{-}Broadcast(\mathcal{P}_i, \{a_{i,j}, b_{i,j}\}, Corrupt)$.
   If the broadcast fails, with $\mathcal{P}_i$ being identified as corrupt, then the receiving players interpret the broadcast values as $a_{i,j} = b_{i,j} = 0$.

Round 3
7. For all $\mathcal{P}_i, \mathcal{P}_j$ for which $a_{i,j} \neq b_{j,i}$ execute:
   (a) Player $\mathcal{P}_k$ executes $Synch\text{-}Broadcast(\mathcal{P}_k, \{F(\alpha_j, \alpha_i)\}, Corrupt)$.
   (b) Player $\mathcal{P}_i$ executes $Synch\text{-}Broadcast(\mathcal{P}_i, \{F_i(\alpha_j)\}, Corrupt)$.
   (c) Player $\mathcal{P}_j$ executes $Synch\text{-}Broadcast(\mathcal{P}_j, \{G_j(\alpha_i)\}, Corrupt)$.
   Again, if any broadcast fails we interpret the result as being the value zero being broad–cast.

Local Computation
8. Set $UnHappy = \emptyset$.
9. For every $\mathcal{P}_i, \mathcal{P}_j$ for which $a_{i,j} \neq b_{j,i}$, where $\mathcal{P}_k$ broadcast $F(\alpha_j, \alpha_i)$, $\mathcal{P}_i$ broadcast $F_i(\alpha_j)$, and $\mathcal{P}_j$ broadcast $G_j(\alpha_i)$:
   (a) If $F_i(\alpha_j) \neq F(\alpha_j, \alpha_i)$ then add $\mathcal{P}_i$ into $UnHappy$.
   (b) If $G_j(\alpha_i) \neq F(\alpha_j, \alpha_i)$ then add $\mathcal{P}_j$ into $UnHappy$.
10. If $|UnHappy| > \mathfrak{t}$ then add $\mathcal{P}_k$ into $Corrupt$ and output the trivial zero sharing $\langle 0 \rangle$.

Round 4
11. For every $\mathcal{P}_i \in UnHappy$ and $\mathcal{P}_j \notin UnHappy$ execute:
    (a) Player $\mathcal{P}_k$ executes $Synch\text{-}Broadcast(\mathcal{P}_k, \{F_i(X)\}, Corrupt)$.
    (b) Player $\mathcal{P}_j$ executes $Synch\text{-}Broadcast(\mathcal{P}_j, \{G_j(\alpha_i)\}, Corrupt)$.
    Again, if any broadcast fails we interpret the result as being the value zero being broad–cast.

Output
12. If there exists $\mathcal{P}_i \in UnHappy$ for which $\mathcal{P}_k$ broadcast $F_i(X)$, and there exists $2 \cdot \mathfrak{t}$ parties $\mathcal{P}_j \notin UnHappy$ which broadcast $G_j(\alpha_i)$, where $F_i(\alpha_j) \neq G_j(\alpha_i)$, then add $\mathcal{P}_k$ into $Corrupt$ and output the trivial zero sharing $\langle 0 \rangle$.
13. Return the sharing of s given by $\langle s \rangle_i = F_i(0)$.

Figure 46: Verifiable Secret Sharing. This is used for protocols in for the *nLarge* threshold profiles..

Recall we are working in the synchronous network model, and we require robustness. This means that an honest player will always respond, and it's message will arrive within time–step Δ of the round. However, a dishonest player may not only try to disrupt the protocol by sending garbage, but it may also try to disrupt the protocol by sending nothing.

We note that our main usage of the VSS is when we want to evaluate the VSS $\mathfrak{n}$–times in parallel, with each party entering a single value into the VSS to be shared (see for example the *CoinFlip* protocol in Figure 54). Thus when implemented, this parallel execution should be taken into account in order to avoid unnecessary rounds. Our Rust implementation batches the protocol by executing the protocol in parallel with each party acting as a sender and merging the communication rounds when possible. Hence, we execute the protocol as written in parallel, and the UC security of this protocol gives us that this parallel composition is indeed UC secure.

An alternative to a parallel implementation is to perform VSS on a batch of secrets instead of only one. The primary change in this implementation is that if inconsistencies are detected for one element in the batch, then the responsible party is flagged even if other elements in the batch pass the verification. For example, the verification in line 9a of Figure 46 will run for batch size B times, i.e., $\{F_i^k(\alpha_j) \neq F^k(\alpha_j, \alpha_i) : k \in [B]\}$, and if one failure occurs out of B, then $\mathcal{P}_i$ is added into *UnHappy*.

7.2.5   Agree on a Random Number

At various points a subset $\mathcal{S} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$ of our set of parties need to agree on a random value in $\{0, 1\}^k$, often the key for a Pseudo–Random Function (PRF) or Pseudo–Random Generator (PRG), which is outside the control of all parties. A simple passively secure protocol for this is given by protocol *AgreeRandom* in Figure 47. Note, the protocol does not verify that the players all obtain the same random value. The idea being that if the same random value is not obtained then this should become apparent once the value is used in a PRF. Thus this method is inherently non–robust, as it will create an invalid execution, and hence an abort if a player deviates from the protocol, but only in a latter protocol which uses the output from *AgreeRandom*. It will also fail to terminate if a player in $\mathcal{S}$ fails to send any required message.

Protocol *AgreeRandom*$(\mathcal{S}, k)$

The input is a subset $\mathcal{S} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$ and a value $k \in \mathbb{N}$ which is the length of the output. Let I denote the index set of the players in $\mathcal{S}$, i.e. $\mathcal{S} = \{\mathcal{P}_i\}_{i \in I}$.
1. Each $\mathcal{P}_i \in \mathcal{S}$ generates $s_i \leftarrow \{0, 1\}^k$.
2. Each $\mathcal{P}_i \in \mathcal{S}$ executes $(c_i, o_i) \leftarrow Commit(s_i, i, sid, rid)$.
3. Each $\mathcal{P}_i \in \mathcal{S}$ sends c_i to all other players in $\mathcal{S}$.
4. When $\mathcal{P}_i$ has received c_j from all $j \in I \setminus \{i\}$ it sends (s_i, o_i) to all other players in $\mathcal{S}$.
5. Each $\mathcal{P}_i \in \mathcal{S}$ executes $v_j \leftarrow Verify(c_j, o_j, s_j, j, sid, rid)$ for all $j \in I \setminus \{i\}$.
6. If any $v_j = false$ then *abort*.
7. Each $\mathcal{P}_i \in \mathcal{S}$ outputs $s \leftarrow \bigoplus_{j \in I} s_j$.

Figure 47: Protocol *AgreeRandom*.

As a halfway house between the above protocol, and one which is fully robust we offer an alternative which is actively secure with abort, namely the protocol *AgreeRandom-w-Abort* in Figure 48. In this version the players confirm that they have indeed obtained the same random value; thus if the protocol terminates (and does not abort) then the parties can proceed knowing the value is the same amongst them all.

The final variant we have is a robust version of *AgreeRandom*, called *AgreeRandom-Robust* and given in Figure 49, which consumes a valid secret shared random value $\langle r \rangle$. This function uses a hash function $\mathcal{H}_{AR}$ which maps the Galois Ring $GR(q, F)$ into the set of bit strings of length k, i.e. $\{0, 1\}^k$. We implement $\mathcal{H}_{AR}$ as

$$\mathcal{H}_{AR}(\alpha) = SHAKE\text{-}256(DSep(AR)\|GRencode(\alpha), k). \tag{32}$$

The addition of the initial two bytes is for domain separation. Note, with this byte wise encoding the input to the hash function $SHAKE\text{-}256$ in this case has byte size $2 + \mathfrak{d} \cdot \lceil q/8 \rceil$.

The input is a subset $\mathcal{S} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$ and a value $k \in \mathbb{N}$ which is the length of the output.
1. All parties in $\mathcal{S}$ execute $s \leftarrow$ *AgreeRandom*($\mathcal{S}, k$).
2. Each $\mathcal{P}_i \in \mathcal{S}$ sends s **privately** to all other players in S.
3. If the value received s_j from player $\mathcal{P}_j$ is such that $s_j = \bot$ or $s_j \neq s$ then *abort*.
4. Each $\mathcal{P}_i \in \mathcal{S}$ outputs s to themselves.

Figure 48: Protocol *AgreeRandom-w-Abort*.

The assumptions on the input value (resp. share) r (resp. $\langle r \rangle$) are that
- The random value r is a uniformly random element in $GR(q, F)$ which the adversary has no control over.
- The secret sharing $\langle r \rangle$ is assumed to be a valid degree $\mathfrak{t}$ sharing of r.
- The random value r is assumed to contain enough entropy, i.e. that

$$q^{\mathfrak{d}} > 2^k.$$

In our application we will set $k = sec$, and this constraint will be satisfied by Parameter Choice 4.

Protocol *AgreeRandom-Robust*($\mathcal{S}, k, \langle r \rangle$)

The input is a subset $\mathcal{S} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$ and a value $k \in \mathbb{N}$ which is the length of the output, and $\langle r \rangle$ a valid random degree $\mathfrak{t}$ sharing amongst the full set of $\mathfrak{n}$ players.
1. Execute *RobustOpen*($\mathcal{S}, \langle r \rangle$).
2. All parties in $\mathcal{S}$ execute $s \leftarrow \mathcal{H}_{AR}(r)$.
3. Each $\mathcal{P}_i \in \mathcal{S}$ outputs s.

Figure 49: Protocol *AgreeRandom-Robust*.

The protocol *AgreeRandom-Robust* works since $\langle r \rangle$ is known on input to be a valid random sharing between all $\mathfrak{n}$. Thus the *RobustOpen* procedure, even though it is towards the players in $\mathcal{S}$, which may contain only one honest player, will definitely generate the correct value for the honest players. This is because the sending $\mathfrak{n}$ players contain at most $\mathfrak{t} < \mathfrak{n}/3$ adversaries.

7.2.6 Generating Random Shamir Secret Sharings

As explained earlier in Section 4.6 we use *PRSS* and *PRZS* methods in threshold profile *nSmall*. As explained earlier, these are very old techniques which go back to [CDI05]. The secret shared output of the *PRSS/PRZS* operation in this section is an element in the full Galois Ring (in the case of a *PRZS* it is the zero element), whereas the secret shared output of the *PRSS-Mask.Next* operation is a "small" element in $\mathbb{Z}/(q)$.

7.2.6.1 PRSS Initialization: The algorithm for initializing all the instances of *PRSS* and *PRZS* we use in threshold profile *nSmall* is given in Figure 50. We use a single initialization routine for all of our *PRSS*–style applications (i.e. not only for *PRSS* but also for the *PRZS* and the *PRSS-Mask* functions below).

The algorithm *PRSS.Init*() iterates over all sets A of size $\mathfrak{n} - \mathfrak{t}$. Thus the complexity of *PRSS.Init*(), i.e. the number of sets A we need to deal with, depends on $\binom{\mathfrak{n}}{\mathfrak{t}}$, which can become very large for large $\mathfrak{n}$ and $\mathfrak{t}$. This is why we restrict the size of $\binom{\mathfrak{n}}{\mathfrak{t}}$ when using such a *PRSS*.

The *PRSS.Init*() algorithm comes in two variants. The first is a non–robust variant which is only active–with–abort secure; i.e. it could output an *abort* if the parties do not agree on the correct values.

However, if it does terminate then the keys are known to be distributed correctly and are in agreement.

The second variant is a fully robust version. The fully robust version makes use of the fact that the value

$$c = \frac{\binom{\mathfrak{n}}{\mathfrak{t}}}{\mathfrak{n} - \mathfrak{t}}$$

is relatively small for threshold profile *nSmall* (between 1.3 and 500). This means that we only need to execute the full blown *VSS* protocol, from Figure 46, a this small number of times in order to generate enough random shared values. This is indeed the only place were we execute the *VSS* protocol for threshold profile *nSmall*.

The *VSS* algorithm is executed $\lceil c \rceil$ times in order to generate random values, then $\binom{\mathfrak{n}}{\mathfrak{t}}$ random values are extracted (using the methods from Section 7.1.3.3). Finally, the random values are then assigned to each set $\mathcal{S}$, and the associated random key is extracted using a private robust opening within the set $\mathcal{S}$.

Implementation Note: In an implementation one may want to execute a single call to *PRSS.Init*() but then utilize the obtained *PRSS/PRZS/PRSS-Mask* in multiple different concurrent executions. We refer to each concurrent executions as a *session*, to which we associate a unique and pseudo–random session–id *sid*. In the case of distributed decryption of an FHE ciphertext, this *sid* could be derived by hashing the input ciphertext for example. In order to avoid conflicts in accessing the counters from multiple sessions, which happen in parallel and can start in different order on different parties, we utilize *per session* counters, but then modify the values r_A output from *PRSS.Init*() to $r_A \oplus sid$ in each session. This implementation modification to cope with out–of–order execution on the different parties is hidden in this design document in order to make the overall exposition easier to follow.

7.2.6.2 PRSS: The algorithms for using the *PRSS* are defined in Figure 51. The *PRSS* makes use of a PRF ψ of the form

$$\psi : \begin{cases} \{0,1\}^{sec} \times S & \longrightarrow & GR(q,F) \\ (\kappa, cnt) & \longmapsto & \psi(\kappa, cnt) \end{cases}$$

where $\{0,1\}^{sec}$ is the key space and S is a set of counters. Note, the shared value which is output by the *PRSS.Next*() invocation is the sharing of the value

$$s \leftarrow \sum_A \psi(r_A, cnt).$$

We now describe how the function ψ is implemented in practice. Since $GR(q,F)(\mathbb{Z}/(q))[X]/F(X)$ we define ψ as a sum of $\mathfrak{d}$ different PRF evaluations

$$\psi(\kappa, cnt) = \sum_{i=0}^{\mathfrak{d}-1} \psi^{(i)}(\kappa, cnt) \cdot X^i,$$

where $\psi^{(i)}$ is a PRF of the form

$$\psi^{(i)} : \begin{cases} \{0,1\}^{sec} \times S & \longrightarrow & \mathbb{Z}/(q) \\ (\kappa, cnt) & \longmapsto & \psi^{(i)}(\kappa, cnt) \end{cases} .$$

One can then implement the $\psi^{(i)}$ using AES, in an obvious counter mode. We let

$$v = \begin{cases} \lceil (dist + \log_2 q)/128 \rceil & q \text{ not a power of two,} \\ \lceil (\log_2 q)/128 \rceil & q \text{ a power of two,} \end{cases}$$

and then define

$$\psi^{(i)}(\kappa, cnt) = \left(AES_\kappa(0\|i\|cnt) + 2^{128} \cdot AES_\kappa(1\|i\|cnt) + 2^{256} \cdot AES_\kappa(2\|i\|cnt) + \dots \right.$$

PRSS.Init(): This is the version which is active–with–abort secure.

    1. For every set $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$:

        (a) $S \leftarrow \{\mathcal{P}_i\}_{i \in A}$.

        (b) Players $\mathcal{P}_i$ with $i \in A$ store $r_A \leftarrow$ *AgreeRandom-w-Abort*(S, sec). The parties in S should execute this protocol over **private** channels, as they wish to keep the value r_A from the other players.

        (c) Define $f_A(X) \in (\mathbb{Z}/(q))[X]$ to be the polynomial of degree $\mathfrak{t}$ such that $f_A(0) = 1$ and $f_A(\alpha_i) = 0$ for all $i \notin A$. Each party $\mathcal{P}_i$ only needs store $f_A(\alpha_i)$ though.

    2. $cnt_{PRSS} \leftarrow 0$.

    3. $cnt_{PRZS} \leftarrow 0$.

    4. $cnt_{Mask} \leftarrow 0$.

PRSS.Init(*Corrupt*): This is the version which is robustly secure.

    1. Let $c = \lceil \binom{\mathfrak{n}}{\mathfrak{t}}/(\mathfrak{n} - \mathfrak{t}) \rceil$.

    2. Each player $\mathcal{P}_i$ generates a random value $s_{i,j} \in GR(q, F)$ for $j = 1, \ldots, c$.

    3. In parallel each player enters $s_{i,j}$ into the system by all parties executing $VSS(\mathcal{P}_i, s_{i,j}, \mathfrak{t}, Corrupt)$ for $i = 1, \ldots, \mathfrak{n}$ and $j = 1, \ldots, c$, so all parties obtain $\langle s_{i,j} \rangle$.
If the player $\mathcal{P}_i$ is already identified as corrupt then the players can ignore their input and simply set $\langle s_{i,j} \rangle$ to be a trivial sharing of the value zero for all values j.

    4. Write $\langle \mathbf{s}_j \rangle = (\langle s_{1,j} \rangle, \ldots, \langle s_{\mathfrak{n},j} \rangle)^\top$ for $j = 1, \ldots, c$.

    5. Extract $c \cdot (\mathfrak{n} - \mathfrak{t})$ random shared values $\langle \mathbf{t}_j \rangle = (\langle t_{j,1} \rangle, \ldots, \langle t_{j,\mathfrak{n}-\mathfrak{t}} \rangle)^\top$ by executing locally

$$\langle \mathbf{t}_j \rangle = M_{\mathfrak{n}, \mathfrak{n}-\mathfrak{t}}^\top \cdot \langle \mathbf{s}_j \rangle.$$

    6. Write $R = \{\langle t_{j,i} \rangle\}$, for $j = 1, \ldots, c$ and $i = 1, \ldots, (\mathfrak{n} - \mathfrak{t})$, so R is a set of $c \cdot (\mathfrak{n} - \mathfrak{t})$ random shared values.

    7. For every set $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$:

        (a) $S \leftarrow \{\mathcal{P}_i\}_{i \in A}$.

        (b) Take the next random sharing $\langle r \rangle$ from the list R, and delete it from R so it is not used again.

        (c) All players $\mathcal{P}_i$ execute *AgreeRandom-Robust*$(S, sec, \langle r \rangle)$ with $\mathcal{P}_i$ storing resulting r_A if $i \in A$.

        (d) Define $f_A(X) \in (\mathbb{Z}/(q))[X]$ to be the polynomial of degree $\mathfrak{t}$ such that $f_A(0) = 1$ and $f_A(\alpha_i) = 0$ for all $i \notin A$. Each party $\mathcal{P}_i$ only needs store $f_A(\alpha_i)$ though.

    8. $cnt_{PRSS} \leftarrow 0$.

    9. $cnt_{PRZS} \leftarrow 0$.

    10. $cnt_{Mask} \leftarrow 0$.

PRSS.get-counters(): Access function.

    1. Return $(cnt_{PRSS}, cnt_{PRZS}, cnt_{Mask})$.

Figure 50: Initialization of the Non–Interactive Pseudo–Random Secret Sharing. This is used for protocols in for the *nSmall* threshold profiles. We give two variants, one active–with–abort secure, and one robust..

$$+ 2^{128 \cdot (v-1)} \cdot AES_K((v-1)\|i\|cnt) \Big) \pmod{q},$$

where we treat the output block of the AES cipher as an integer in $[0, \ldots, 2^{128} - 1]$. The input value is made up of an index in $[0, \ldots, (v-1)]$ padded to eight bits, an index i also padded to eight bits, and the *PRSS* counter *cnt* padded to 112 bits. Note, the output of $\psi^{(i)}$ is only statistically uniform in $\mathbb{Z}/(q)$ with distance bounded by 2^{-dist}.

    To obtain robustness for the offline phase of the MPC protocol below when $\mathfrak{t} < \mathfrak{n}/3$, we need to add an extra command into our non–interactive *PRSS* functionality. This extra command allows us to recompute all the shares from a given execution of the *PRSS*, this enables all parties to check in our

PRSS.Next():
    1. Party $\mathcal{P}_i$ computes, where the sum is over every set $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$ containing i,
$$\langle s \rangle_i \leftarrow \sum_{A : i \in A} \psi(r_A, cnt_{PRSS}) \cdot f_A(\alpha_i).$$
    2. $cnt_{PRSS} \leftarrow cnt_{PRSS} + 1$.
    3. Return $\langle s \rangle$.

PRSS.Check(*cnt*, *Corrupt*):
    1. Each party $\mathcal{P}_i \notin Corrupt$ computes the values $\psi(r_A, cnt)$ for all sets $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$ such that $i \in A$.
    2. Each party $\mathcal{P}_i \notin Corrupt$ broadcasts the values $\Psi_i = \{\psi_{A,i} = \psi(r_A, cnt)\}_{\forall A \text{ s.t. } i \in A}$ using *Synch-Broadcast*($\mathcal{P}_i, \Psi_i, Corrupt$) to all other players.
    3. Each party $\mathcal{P}_i \notin Corrupt$ on receiving the sets $\Psi_j = \{\psi_{A,j}\}$ for all j, takes as the true value of $\psi_A = \psi(r_A, cnt)$, for each set A, the majority value. Note, even if one of the broadcasts returned $\bot$ the agreed ψ_A will still be correct.
    4. For any j and A such that $j \in A$ and $\psi_A \neq \psi_{A,j}$, add player $\mathcal{P}_j$ to *Corrupt*.
    5. Party $\mathcal{P}_i \notin Corrupt$ computes for all j the value
$$\langle s \rangle_j \leftarrow \sum_{A : j \in A} \psi(r_A, cnt) \cdot f_A(\alpha_j).$$
    6. Party $\mathcal{P}_i \notin Corrupt$ outputs $(\langle s \rangle_1, \ldots, \langle s \rangle_{\mathfrak{n}})$.

Figure 51: Non–Interactive Pseudo–Random Secret Sharing. This is used for protocols in for the *nSmall* threshold profiles..

protocol who cheated in an execution. The protocol makes use of a broadcast step. This is important to ensure correctness, if we just used point–to–point communication we would need a more complex dispute control framework below. However, this broadcast will only used on an unhappy path (when we know we need to call *PRSS.Check*), and thus only when an error is detected. Thus we sacrifice an expensive broadcast on the unhappy path, for a more simple protocol description and implementation.

7.2.6.3 PRZS: In the threshold profile *nSmall* we also require a method to generate, in a non–interactive manner, a degree $2 \cdot \mathfrak{t}$ sharing of zero, a so called *PRZS*. This is done in a standard manner using Figure 52, which also includes a checking procedure as above. The *PRZS* utilizes the same initialization procedure as that for *PRSS*. This procedure makes use of a PRF χ of the form

$$\chi : \begin{cases} \{0,1\}^{sec} \times S \times \{1, \ldots, t\} & \longrightarrow & GR(q, F) \\ (\kappa, cnt, j) & \longmapsto & \chi(\kappa, cnt, j) \end{cases}$$

where $\{0,1\}^{sec}$ is the keyspace and S is a set of counters.

One can implement χ much like we did ψ above. We first write χ as the sum of $\mathfrak{d}$ different PRF evaluations

$$\chi(\kappa, cnt, j) = \sum_{i=0}^{\mathfrak{d}-1} \chi^{(i)}(\kappa, cnt, j) \cdot X^i,$$

where $\chi^{(i)}$ is a PRF of the form

$$\chi^{(i)} : \begin{cases} \{0,1\}^{sec} \times S \times \{1, \ldots, t\} & \longrightarrow & \mathbb{Z}/(q) \\ (\kappa, cnt, j) & \longmapsto & \chi^{(i)}(\kappa, cnt, j) \end{cases}.$$

One can then implement the $\chi^{(i)}$ much like we implemented $\psi^{(i)}$ above. We let, as before,

$$v = \begin{cases} \lceil (dist + \log_2 q)/128 \rceil & q \text{ not a power of two,} \\ \lceil (\log_2 q)/128 \rceil & q \text{ a power of two} \end{cases}$$

and then define

$$\chi^{(i)}(\kappa, cnt, j) = \Big(AES_{\kappa \oplus 1}(0\|i\|j\|cnt) + 2^{128} \cdot AES_{\kappa \oplus 1}(1\|i\|j\|cnt) + 2^{256} \cdot AES_{\kappa \oplus 1}(2\|i\|j\|cnt) + \ldots$$
$$+ 2^{128 \cdot (v-1)} \cdot AES_{\kappa \oplus 1}((v-1)\|i\|j\|cnt) \Big) \pmod{q},$$

where we treat the output block of the AES cipher as an integer in $[0, \ldots, 2^{128} - 1]$. As inputs, we pad with zeros the indexes of the block v, i and j to eight bits, and cnt is padded to $128 - 3 \cdot 8 = 104$ bits. Notice, the use of $\kappa \oplus 1$ in the key of the AES function, this is to ensure domain separation as we use the same r_A values (i.e. keys) for both the *PRSS* and the *PRZS* operations.

PRZS

PRZS.Next():
1. Party $\mathcal{P}_i$ computes, where the sum is over every set $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$ containing i,

$$\langle z \rangle_i^{2 \cdot \mathfrak{t}} \leftarrow \sum_{A : i \in A} \left(\sum_{j=1}^{\mathfrak{t}} \chi(r_A, cnt_{PRZS}, j) \cdot \alpha_i^j \cdot f_A(\alpha_i) \right).$$

2. $cnt_{PRZS} \leftarrow cnt_{PRZS} + 1$.
3. Return $\langle z \rangle^{2 \cdot \mathfrak{t}}$.

PRZS.Check($cnt, Corrupt$):
1. Each party $\mathcal{P}_i \notin Corrupt$ computes the values $\chi(r_A, cnt, j)$ for all sets $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$ such that $i \in A$, and all $j \in [1, \ldots, \mathfrak{t}]$.
2. Each party $\mathcal{P}_i \notin Corrupt$ broadcasts the values $X_i = \{\chi_{A,j,i} = \chi(r_A, cnt, j)\}_{\forall A \text{ s.t. } i \in A, j \in [1, \ldots, \mathfrak{t}]}$ using *Synch-Broadcast*($\mathcal{P}_i, X_i, Corrupt$) to all other players.
3. Each party $\mathcal{P}_i \notin Corrupt$ on receiving the sets $X_k = \{\chi_{A,j,k}\}$ for all k, takes as the value of $\chi_{A,j}$, for each set A and index j, the majority value. Note, if the broadcast returned $\perp$ above then $\chi_{A,j}$ will still be correct.
4. If for any k, A and j we have $\chi_{A,j} \neq \chi_{A,j,k}$ then add player $\mathcal{P}_k$ to *Corrupt*.
5. Party $\mathcal{P}_i \notin Corrupt$ computes for all k the value

$$\langle z \rangle_k^{2 \cdot \mathfrak{t}} \leftarrow \sum_{A : k \in A} \left(\sum_{j=1}^{\mathfrak{t}} \chi(r_A, cnt_{PRZS}, j) \cdot \alpha_k^j \cdot f_A(\alpha_k) \right).$$

6. Party $\mathcal{P}_i \notin Corrupt$ outputs $(\langle z \rangle_1^{2 \cdot \mathfrak{t}}, \ldots, \langle z \rangle_{\mathfrak{n}}^{2 \cdot \mathfrak{t}})$.

Figure 52: Non–Interactive Pseudo–Random Zero Sharing. This is used for protocols in for the *nSmall* threshold profiles..

7.2.6.4 PRSS–Mask:

The *PRSS-Mask* operations makes use of a PRF ϕ of the form

$$\phi : \begin{cases} \{0,1\}^{sec} \times S & \longrightarrow & \mathbb{Z} \\ (\kappa, cnt) & \longmapsto & \phi(\kappa, cnt) \end{cases}$$

where $\{0,1\}^{sec}$ is the keyspace and S is a set of counters. The output of the function ϕ is assumed to be bounded in absolute value by $Bd_1 = 2^{stat} \cdot Bd$. Again, one can implement ϕ using AES, again as we

did ψ and χ above, with again using a key tweak (this time of $\oplus 2$ to ensure domain separation). We let, as before,

$$v = \begin{cases} \lceil (dist + \log_2(2 \cdot Bd_1))/128 \rceil & Bd_1 \text{ not a power of two,} \\ \lceil (\log_2(2 \cdot Bd_1))/128 \rceil & Bd_1 \text{ a power of two} \end{cases}$$

and then define

$$\phi^{Bd_1}(\kappa, cnt) = -Bd_1 + \Big(\big(AES_{\kappa \oplus 2}(0 \| cnt) + 2^{128} \cdot AES_{\kappa \oplus 2}(1 \| cnt) + 2^{256} \cdot AES_{\kappa \oplus 2}(2 \| cnt) + \ldots$$
$$+ 2^{128 \cdot (v-1)} \cdot AES_{\kappa \oplus 2}((v-1) \| cnt) \big) \pmod{2 \cdot Bd_1}, \Big)$$

where we treat the output block of the AES cipher as an integer in $[0, \ldots, 2^{128} - 1]$. As input, we have as before an index in $[0, \ldots, (v-1)]$ padded to eight bits, and the input value cnt to the AES function padded to 120 bits. Note, the output of ψ is only statistically uniform in $[-Bd_1, \ldots, Bd_1]$ with distance bounded by 2^{-stat}.

The shared value which is output by the *PRSS-Mask.Next* invocation is the sharing of the value

$$E \leftarrow \sum_A (\phi(r_A, cnt) + \phi(r_A, cnt + 1)).$$

Since the output of ϕ is bounded as above, we have that the value E, whose sharing is output by Figure 53, is bounded in absolute value by

$$2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot Bd_1 = 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \cdot Bd,$$

as the sum used in the *PRSS* has at most $2 \cdot \binom{n}{t}$ terms.

PRSS–Mask

PRSS-Mask.Next($Bd, stat$):
    1. Set $Bd_1 = 2^{stat} \cdot Bd$.
    2. Party $\mathcal{P}_i$ computes, where the sum is over every set A containing i,

$$\langle E \rangle_i \leftarrow \sum_{A: i \in A} \Big(\phi^{Bd_1}(r_A, cnt_{Mask}) + \phi^{Bd_1}(r_A, cnt_{Mask} + 1) \Big) \cdot f_A(\alpha_i).$$

    3. $cnt_{Mask} \leftarrow cnt_{Mask} + 2$.
    4. Return $\langle E \rangle$.

Figure 53: Pseudo–Random Secret Sharing *PRSS-Mask* for use in Threshold Profiles in Category *nSmall*.

7.2.7 CoinFlip

Given the VSS protocol from Figure 46 we can define a secure *CoinFlip* protocol as given in Figure 54. This is the only place where we actually use the full VSS protocol in threshold profile *nLarge*. We use it to generate a single random element in $GR(q, F)$.

CoinFlip(*Corrupt*):
1. All players $\mathcal{P}_i \notin Corrupt$ pick a random value $x_i \in GR(q, F)$ and call *VSS*($\mathcal{P}_i, x_i, \mathfrak{t}, Corrupt$), so that all players obtain $\langle x_i \rangle$.
2. The players locally compute $\langle x \rangle = \sum_{\mathcal{P}_i \notin Corrupt} \langle x_i \rangle$.
3. The players execute $x \leftarrow RobustOpen(\{\mathcal{P}_1, \ldots, \mathcal{P}_\mathfrak{n}\}, \langle x \rangle)$ and output x.

Figure 54: Coin Flip Protocol. This is used for protocols in for the *nLarge* threshold profiles..

7.3 Layer Two

We summarize the relationship between the protocols in Layer Two in Figure 55 and Figure 56.
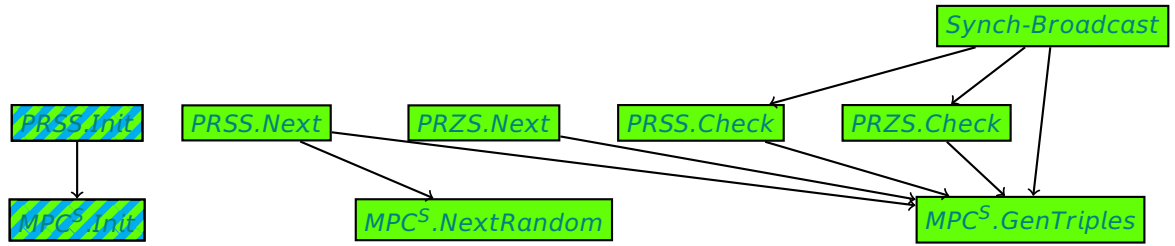
Figure 55: Offline MPC Protocols for Threshold Profile Category *nSmall*. These protocols assume **synchronous** networks. .
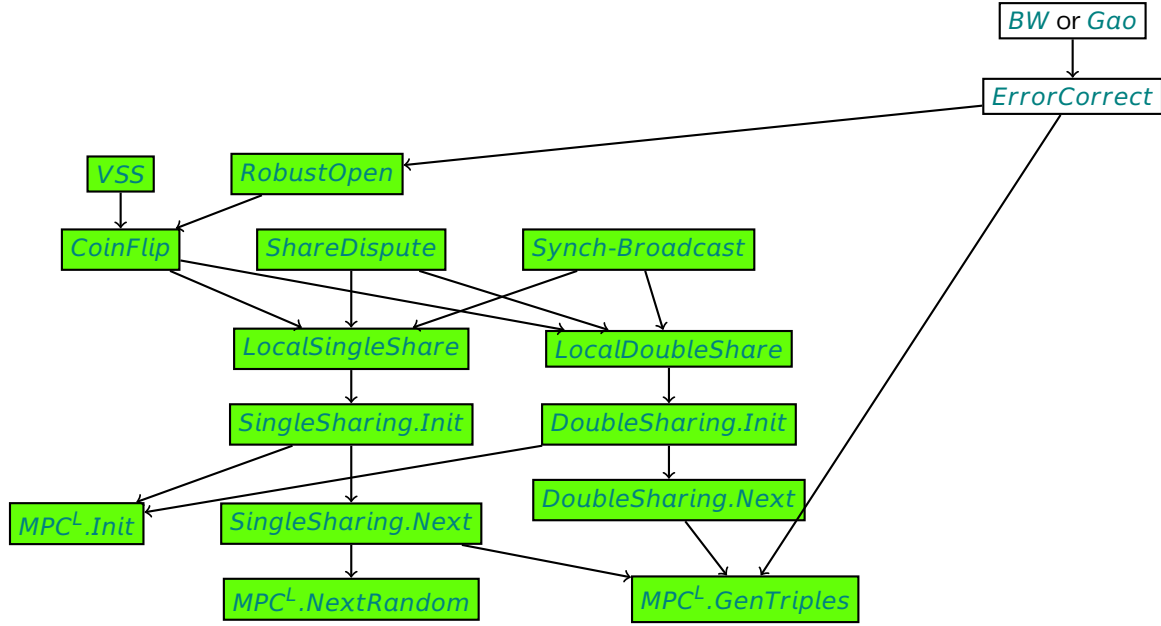
Figure 56: Offline MPC Protocols for Threshold Profile Category *nLarge*. These protocols assume **synchronous** networks.

7.3.1   Offline MPC Protocol for Threshold Profile Category *nSmall*

We now have all the basics needed to describe the offline MPC protocols for threshold profile category *nSmall*. The initialization and triple generation methods are given in Figure 57.

The generation of multiplication triples is done using the Damgård–Nielsen multiplication protocol. Unlike in [DN07] we do not use batching/the "king" paradigm. As mentioned earlier, this is because the "king" methodology, whilst decreasing communication complexity, increases round complexity. Thus the number of players would have to be very large for this to produce a major effect on performance when implemented over WANs. See Section 7.2.3.1 for how to perform batching in this context. Applying this optimization we leave for the reader to investigate in their specific implementation context.

To deal with the unhappy path we utilize our novel checking procedure, which utilizes two features of our protocol. Firstly, the initial shares are generated using a *PRSS*, and thus can be regenerated by everyone using the *PRSS.Check* function. Secondly, the opening in the Damgård–Nielsen multiplication is opened via a broadcast, and thus each player is committed to their message to all other players. We note that this simplification appears to be novel, we could not find it in the existing literature, thus we shall show this is secure in Section 8.3.1.

We assume that the broadcast communication in Figure 57 line 1h, is executed in parallel (so that the work to authenticate the broadcasts are performed on many triples at once, rather than executed per triple). This parallel execution is ignored in the protocol description for ease of exposition.

The broadcast could be amortized over many executions if one is willing to assume digital signatures, and not just authentic channels. For this optimization one exchanges the shares of the $\langle d \rangle_i^{2 \cdot t}$ values for ℓ values of d using pair–wise channels and signing the messages. Then the parties proceed, and execute a single broadcast to signal if they are happy or not. In the case of one unhappy player the ℓ signed messages are exchanged, the signatures verified, and the checking procedure is executed as stated. On the happy path this means that the expensive broadcast is amortized over ℓ triples, however it comes at the expense of a more complex processing on the unhappy path, and the need for digital signatures. This "optimization" is not considered further in this document.

In Figure 57 we also define the function $MPC^S.NextRandom(Dispute)$, which is simply an alias for

Note, this assumes $\mathfrak{t} < \mathfrak{n}/3$, and synchronous networks.

$MPC^S.Init()$:
1. Call $PRSS.Init()$ or $PRSS.Init(Corrupt)$ depending on whether active–with–abort or robust security is required for $MPC^S.Init()$.
2. Define $\mathcal{T}, \mathcal{B} \leftarrow \emptyset$. The set $\mathcal{T}$ is the set of multiplication triples generated, whilst the set $\mathcal{B}$ is the set of shared random bits generated.

$MPC^S.GenTriples(Dispute)$:
1. Execute the following a sufficient number of times in parallel in order to generate enough triples for the online phase:
   (a) $(cnt_1, cnt_2, cnt_3) \leftarrow PRSS.get\text{-}counters()$.
   (b) $\langle x \rangle \leftarrow PRSS.Next()$.
   (c) $\langle y \rangle \leftarrow PRSS.Next()$.
   (d) $\langle v \rangle \leftarrow PRSS.Next()$.
   (e) $\langle z \rangle^{2 \cdot \mathfrak{t}} \leftarrow PRZS.Next()$.
   (f) $\langle v \rangle^{2 \cdot \mathfrak{t}} \leftarrow \langle v \rangle + \langle z \rangle^{2 \cdot \mathfrak{t}}$.
   (g) Locally compute for all $\mathcal{P}_i$ the share $\langle d \rangle_i^{2 \cdot \mathfrak{t}} \leftarrow \langle x \rangle_i \cdot \langle y \rangle_i + \langle v \rangle_i^{2 \cdot \mathfrak{t}}$.
   (h) All players $\mathcal{P}_i \notin Corrupt$ execute $Synch\text{-}Broadcast(\mathcal{P}_i, \langle d \rangle_i^{2 \cdot \mathfrak{t}}, Corrupt)$. If $\mathcal{P}_j$ is added to $Corrupt$ by this call to $Synch\text{-}Broadcast$, or $\mathcal{P}_j$ was in $Corrupt$ before the call, then the assumed broadcast value is assumed to be $\langle d \rangle_j^{2 \cdot \mathfrak{t}} = \perp$.
   (i) All players $\mathcal{P}_i \notin Corrupt$ therefore have an agreed set of shares $\{\langle d \rangle_j^{2 \cdot \mathfrak{t}}\}$ of which $s \leq \mathfrak{t}$ are equal to $\perp$.
   (j) All players $\mathcal{P}_i \notin Corrupt$ apply the error correction algorithm $G(X) \leftarrow ErrorCorrect(q, (\langle d \rangle_j^{2 \cdot \mathfrak{t}})_{j=1}^{\mathfrak{n}})$. The algorithm is applied with $\mathfrak{d} = 2 \cdot \mathfrak{t} + 1$, thus it will output $\perp$ only if there is more than $(\mathfrak{n} - s - 2 \cdot \mathfrak{t} - 1)/2 > (\mathfrak{t} - s)/2$ errors in the values $\{\langle d \rangle_j^{2 \cdot \mathfrak{t}}\}$.
   (k) If $G(X) \neq \perp$ then execute the following *happy path*
      i. $d \leftarrow G(0)$.
      ii. $\langle z \rangle \leftarrow d - \langle v \rangle$.
      iii. $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\langle x \rangle, \langle y \rangle, \langle z \rangle)\}$.
   (l) Else execute the following *unhappy path*
      i. $(\langle x \rangle_1, \ldots, \langle x \rangle_{\mathfrak{n}}) \leftarrow PRSS.Check(cnt_1, Corrupt)$.
      ii. $(\langle y \rangle_1, \ldots, \langle y \rangle_{\mathfrak{n}}) \leftarrow PRSS.Check(cnt_1 + 1, Corrupt)$.
      iii. $(\langle v \rangle_1, \ldots, \langle v \rangle_{\mathfrak{n}}) \leftarrow PRSS.Check(cnt_1 + 2, Corrupt)$.
      iv. $(\langle z \rangle_1^{2 \cdot \mathfrak{t}}, \ldots, \langle z \rangle_{\mathfrak{n}}^{2 \cdot \mathfrak{t}}) \leftarrow PRZS.Check(cnt_2, Corrupt)$.
      v. Compute for all i the values
         A. $\langle v \rangle_i^{2 \cdot \mathfrak{t}} \leftarrow \langle v \rangle_i + \langle z \rangle_i^{2 \cdot \mathfrak{t}}$.
         B. $\langle d' \rangle_i^{2 \cdot \mathfrak{t}} \leftarrow \langle x \rangle_i \cdot \langle y \rangle_i + \langle v \rangle_i^{2 \cdot \mathfrak{t}}$.
         C. If $\langle d' \rangle_i^{2 \cdot \mathfrak{t}} \neq \langle d \rangle_i^{2 \cdot \mathfrak{t}}$ then add player $\mathcal{P}_i$ to $Corrupt$.
      vi. Triple production can then resume, but with the players in $Corrupt$ removed from the computation completely, and the triples "produced" from these values of cnt_1 and cnt_2 being thrown away.

$MPC^S.NextRandom(Dispute)$:
1. $\langle x \rangle \leftarrow PRSS.Next()$.
2. Return $\langle x \rangle$.

Figure 57: The Offline Procedures for Threshold Profiles in Category *nSmall*.

PRSS.Next().

7.3.2 Offline MPC Protocol for Threshold Profile Category *nLarge*

This version of the offline protocol works when $\binom{n}{t}$ is large. This means we can no long rely on *PRSS* operations to produce any sharings "for free". To avoid, as much as possible, the usage of the expensive *VSS* protocol we utilize the framework of [DNO7]. What this framework does is provide a "batched" VSS protocol which is statistically secure, i.e. it has a vanishingly small probability of outputting invalid sharings. In the following we set this vanishingly small probability to be 2^{-dist}, i.e. 2^{-80} given our global choice of *dist*. This batched VSS protocol replaces the *PRSS* from above.

The "double sharings" needed to produce the multiplication triples are then produced by a modification of this "batched" VSS protocol. The "batched" VSS protocols take as input sharings from parties, and then combine them using the standard technique of randomness extraction discussed in Section 7.1.3.3.

All the above needs to be accomplished, whilst ensuring that pairs of parties in *Dispute* do not keep signaling the same dispute over and over again. To enable this, following [DNO7] we define a "sharing with disputes" protocol, which ensures that a party only shares input values with parties for which it is not in dispute; see Figure 58.

```
Sharing with Disputes
```

$ShareDispute(\mathcal{P}_i, s, d, Dispute)$: To share a value $s \in R = GR(q, F)$ via a degree d sharing we execute the steps.
   1. $\mathcal{P}_i$ selects a random polynomial $F(X) \in R[X]$ of degree d such that $F(0) = s$ and $F(\alpha_j) = 0$ for all $\mathcal{P}_j$ such that $\{\mathcal{P}_i, \mathcal{P}_j\} \in Dispute$.
   2. Send $\langle s \rangle_j^d = F(\alpha_j)$ **privately** to all player $\mathcal{P}_j$ for which $\{\mathcal{P}_i, \mathcal{P}_j\} \notin Dispute$.
      If $\mathcal{P}_j$ receives nothing, then he assumes that the value sent was $\langle s \rangle_j^d = 0$.
   3. Player $\mathcal{P}_i$ remembers the values $\langle s \rangle_j^d$ for later.

Figure 58: Passive Securely Input a Sharing Disputes. This is used for protocols in for the *nLarge* threshold profiles..

We implement, in practice, a parallel version of the protocol in Figure 58, where each party shares a value with all other parties at he same time, as this is exactly how the protocol is used higher up in our protocol stack. The security of this usage is guaranteed by the UC framework.

7.3.2.1 A "Batched" Statistical VSS:
In this section we present a method to enable a single party to enter a vector of sharings (or double sharings) via a "batched" VSS protocol, see Figure 60 and Figure 61. These protocols, for the finite field case, form the backbone of the MPC protocol from [DNO7]. The extension to the case of Galois Rings is immediate, however for completeness, later, in Theorem 5, we prove correctness and security in general.

The checking of the validity of the ℓ input sharings in these protocols makes use of the Schwartz–Zippel Lemma (Lemma 10). For this lemma we make use of the exceptional sequence $S = GR(p, F)$; which has size $p^{\mathfrak{d}}$. This means we need to repeat the checking a total of m times where

$$\left(\frac{1}{p^{\mathfrak{d}}}\right)^m \leq 2^{-dist},$$

i.e. we set

$$m = \left\lceil \frac{dist}{\log_2(p^{\mathfrak{d}})} \right\rceil. \tag{33}$$

To obtain a challenge we make use of a hash function to map the output of *CoinFlip* to a tuple which we

then use to evaluate the multinomial

$$\mathcal{H}_{LDS} : GR(q, F) \times \{1, \ldots, m\} \times \{1, \ldots, \mathfrak{n}\} \longrightarrow S^{\ell}.$$

This hash function is defined in Figure 59. Since $S = GR(p, F)$ is a finite field, there is a multiplicative generator gen of the set $S \setminus \{0\}$, which we assume is fixed and known to all parties. Note that, in the algorithm Figure 59 when $p^{\mathfrak{d}}$ is not a power of two we generate more data from the application of $SHAKE\text{-}256$, in order to ensure the generated element in $S = GR(p, F)$ is closer to uniformly selected.

$\mathcal{H}_{LDS}(x, g, i)$

$\mathcal{H}_{LDS}(x, g, i)$:
1. $m \leftarrow DSep(LDS)\|GRencode(x)\|g\|i$, where the index g is encoded as a single byte, and the index i is encoded also as a two bytes.
   Hence $0 \leq g \leq 255$ *and* $0 \leq i \leq 65535$.
2. $v \leftarrow 8 \cdot \lceil p^{\mathfrak{d}}/8 \rceil$.
3. If $p \neq 2$ then $v \leftarrow 2 \cdot v$.
4. $(a_1, \ldots, a_{\ell}) \leftarrow SHAKE\text{-}256(m, \ell \cdot d)$.
5. For $i \in [1, \ldots, \ell]$
   (a) $b_i \leftarrow a_i \pmod{p^{\mathfrak{d}}} \in [0, \ldots, p^{\mathfrak{d}} - 1]$.
   (b) If $b_i = p^{\mathfrak{d}} - 1$ then $y_i \leftarrow 0$.
   (c) Else $y_i \leftarrow gen^{b_i}$.
6. Return $(y_1, \ldots, y_{\ell})$.

Figure 59: The Hash Function $\mathcal{H}_{LDS}(x, g, i)$.

Note, the use of the dispute control framework is restricted only to the execution of the protocols *LocalSingleShare* and *LocalDoubleShare*, everything else in our protocol is robust "by default". In addition the only place where we require broadcast, or any form of synchronicity, is also in *LocalSingleShare* and *LocalDoubleShare* (and by implication in all the implemented sub–protocols). Thus we have isolated the main issues re dispute resolution, broadcast and synchronicity to one location.

As with the previous protocols, we implement in our code a parallel version of *LocalSingleShare*, where every party shares their own vector of secrets in parallel. Contrary to previous protocols, here we do not just execute the stated protocol in parallel n times because we use the same *CoinFlip* across the parallel executions. Thus the actual implemented protocol realizes not quite the same UC functionality as just executing the stated protocol in parallel. This modification to the protocol description is easy to cope with in the modeling, and is thus left to the reader. We adopt the more simple presentation here for readability reasons.

Our explanation of *CoinFlip*(*Corrupt*) above has an "optimization" which is not in the presenta–tion in [DN07]. This optimization is that the *CoinFlip*(*Corrupt*) can update the known set of corrupt parties. This means that in protocols *LocalSingleShare* and *LocalDoubleShare*, if the execution of *CoinFlip*(*Corrupt*), on line 4, results in an increase in the set *Corrupt* then we need to return to the start of the protocol to ensure that the execution of *ShareDispute* is consistent with this new set *Corrupt*(*Corrupt*) (and hence the new set *Dispute*).

Note, a similar optimization to that used in the protocol *CoinFlip*(*Corrupt*) is applied also in the protocol *Synch-Broadcast*($\mathcal{P}_i, \cdot, Corrupt$), however in this case one only adds to *Corrupt* if the sender's actions result in an $\perp$ value being broadcast. This would trip the error processing in *LocalSingleShare* and *LocalDoubleShare* to add $\mathcal{P}_i$ to *Corrupt* even if *Synch-Broadcast*($\mathcal{P}_i, \cdot, Corrupt$) did not. Thus for the single call to the protocol *Synch-Broadcast*($\mathcal{P}_i, \cdot, Corrupt$), no modification it appears to be needed. However, if one runs *LocalSingleShare* and *LocalDoubleShare* in the parallel setting, described above, then this update to *Corrupt* can effect the correctness of the other parallel executions. Thus to be safe we execute a return to the first step always if the size of *Corrupt* increases. This return should happen

for all parallel executions.

Local Checked Single Sharing

$LocalSingleShare(\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute)$: For $s_i \in R = GR(q, F)$ chosen by $\mathcal{P}_i$ this procedure goes as follows.

1. If $\mathcal{P}_i \in Corrupt$ then set $s_j = 0$, $\langle s_j \rangle_k = 0$ for $j = 1, \ldots, \ell$ and $k = 1, \ldots, n$, and return.
2. Player $\mathcal{P}_i$ calls $ShareDispute(\mathcal{P}_i, s_j, \mathfrak{t}, Dispute)$ for $j = 1, \ldots, \ell$ to obtain $\langle s_j \rangle$.
3. Player $\mathcal{P}_i$, for $g = 1, \ldots, m$, generates a random $r_g \in R$ and then calls $\langle r_g \rangle \leftarrow ShareDispute(\mathcal{P}_i, r_g, \mathfrak{t}, Dispute)$, where m is defined by equation (33).
4. $x \leftarrow CoinFlip(Corrupt) \in R$.
   When executing this protocol in parallel, this is a single call to CoinFlip and not $\mathfrak{n}$ such calls.
5. If the call to $CoinFlip(Corrupt)$ above resulted in an increase to the set $Corrupt$ then return to the first step for all parallel executions.
6. For $g \in [1, \ldots, m]$ (can be executed in parallel)
   (a) $(x_{1,g}, \ldots, x_{\ell,g}) \leftarrow \mathcal{H}_{LDS}(x, g, i)$, note $x_{i,g} \in S$.
   (b) All parties compute
   $$\langle y_g \rangle \leftarrow \langle r_g \rangle + \sum_{j=1}^{\ell} x_{j,g} \cdot \langle s_j \rangle.$$
   (c) Player $\mathcal{P}_i$, using the values remembered from calling $ShareDispute$, computes the values $\langle y_g \rangle_j$ for all j, call this share $\langle y_g^* \rangle_j$.
   (d) All parties $\mathcal{P}_j$ in $Agree_i$ execute $Synch\text{-}Broadcast(\mathcal{P}_j, \langle y_g \rangle_j, Corrupt)$, by convention all parties in $Dispute_i$ are "defined" to broadcast $\langle y_g \rangle_j = 0$.
   (e) Simultaneously player $\mathcal{P}_i$ executes $Synch\text{-}Broadcast(\mathcal{P}_i, \langle y_g^* \rangle_j, Corrupt)$ for all j.
   (f) If any of the preceding calls to $Synch\text{-}Broadcast(\cdot, \cdot, Corrupt)$ results in an increase to $Corrupt$ then return to step 1 for all parallel executions.
   (g) If the sharing $\langle y_g^* \rangle_j$ broadcast by $\mathcal{P}_i$ is not a degree t sharing with all parties in $Dispute_i$ having the zero share, then add $\mathcal{P}_i$ to $Corrupt$, and return to step 1 for all parallel executions.
   *If this is **not** being run in parallel then one could simply set $s_j = 0$ and $\langle s_j \rangle_k = 0$ for $j = 1, \ldots, \ell$ and $k = 1, \ldots, n$, and not return to step 1.*
   (h) If $\langle y_g \rangle_j \neq \langle y^* \rangle_j$ then add $\{\mathcal{P}_i, \mathcal{P}_j\}$ to $Dispute$ for all $\mathcal{P}_j \in Agree_i$, and return to step 1 for all parallel executions.
   Note all parties will agree on the dispute having happened as they can all do the checks, which are on broadcast values.
7. Return $((\langle s_i \rangle))_{i=1}^{\ell}$.

Figure 60: Locally Produced, by $\mathcal{P}_i$, Checked Single Sharing of Multiple Values with Disputes. This is used for protocols in for the *nLarge* threshold profiles..

Local Checked Double Sharing

$LocalDoubleShare(\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute)$: For $s_i \in R = GR(q, F)$ chosen by $\mathcal{P}_i$ this procedure goes as follows.

1. If $\mathcal{P}_i \in Corrupt$ then set $s_j = 0$, $\langle s_j \rangle_k = 0$ and $\langle s_j \rangle_k^{2 \cdot t} = 0$, for $j = 1, \ldots, \ell$ and $k = 1, \ldots, n$, and return.

2. Player $\mathcal{P}_i$ calls $ShareDispute(\mathcal{P}_i, s_j, t, Dispute)$ for $j = 1, \ldots, \ell$ to obtain $\langle s_j \rangle$.

3. Player $\mathcal{P}_i$ calls $ShareDispute(\mathcal{P}_i, s_j, 2 \cdot t, Dispute)$ for $j = 1, \ldots, \ell$ to obtain $\langle s_j \rangle^{2 \cdot t}$.

4. Player $\mathcal{P}_i$, for $g = 1, \ldots, m$, generates a random $r_g \in R$ and then calls $\langle r_g \rangle \leftarrow ShareDispute(\mathcal{P}_i, r_g, t, Dispute)$ and $\langle r_g \rangle^{2 \cdot t} \leftarrow ShareDispute(\mathcal{P}_i, r_g, 2 \cdot t, Dispute)$, where m is defined by equation (33).

5. $x \leftarrow CoinFlip(Corrupt) \in R$.
   When executing this protocol in parallel, this is a single call to $CoinFlip$ and not $\mathfrak{n}$ such calls.

6. If the call to $CoinFlip(Corrupt)$ above resulted in an increase to the set $Corrupt$ then return to the first step for all parallel executions.

7. For $g \in [1, \ldots, m]$ (can be executed in parallel)
   (a) $(x_{1,g}, \ldots, x_{\ell,g}) \leftarrow \mathcal{H}_{LDS}(x, g, i)$, note $x_{i,g} \in S$.
   (b) All parties compute
   $$\langle y_g \rangle \leftarrow \langle r_g \rangle + \sum_{j=1}^{\ell} x_{j,g} \cdot \langle s_j \rangle$$
   and
   $$\langle y_g \rangle^{2 \cdot t} \leftarrow \langle r_g \rangle^{2 \cdot t} + \sum_{j=1}^{\ell} x_{j,g} \cdot \langle s_j \rangle^{2 \cdot t}.$$

   (c) Player $\mathcal{P}_i$, using the values remembered from calling $ShareDispute$, computes the values $\langle y_g \rangle_j$ and $\langle y_g \rangle_j^{2 \cdot t}$ for all j, call these $\langle y_g^* \rangle_j$ and $\langle y_g^* \rangle_j^{2 \cdot t}$.

   (d) All parties $\mathcal{P}_j$ in $Agree_i$ execute $Synch\text{-}Broadcast(\mathcal{P}_j, \{\langle y_g \rangle_j, \langle y_g \rangle^{2 \cdot t_j}\}, Corrupt)$, whilst all parties in $Dispute_i$ are deemed to have broadcast $\langle y_g \rangle_j = \langle y_g \rangle^{2 \cdot t} = 0$.

   (e) Simultaneously player $\mathcal{P}_i$ executes $Synch\text{-}Broadcast(\mathcal{P}_i, \{\langle y_g^* \rangle_j, \langle y_g^* \rangle_j^{2 \cdot t}\}, Corrupt)$.

   (f) If any of the preceding calls to $Synch\text{-}Broadcast(\cdot, \cdot, Corrupt)$ results in an increase to $Corrupt$ then return to step 1 for all parallel executions.

   (g) If one of the following is true
   - The sharing $\langle y_g^* \rangle_j$ broadcast by $\mathcal{P}_i$ is not a degree t sharing with all parties in $Dispute_i$ having the zero share,
   - The sharing $\langle y_g^* \rangle_j^{2 \cdot t}$ broadcast by $\mathcal{P}_i$ is not a degree $2 \cdot t$ sharing with all parties in $Dispute_i$ having the zero share,
   - Upon reconstructing y_g from both $\langle y \rangle$ and $\langle y \rangle^{2 \cdot t}$ the two values are different
   then add $\mathcal{P}_i$ to $Corrupt$, and return to step 1 for all parallel executions.
   *If this is **not** being run in parallel then one could simply set $s_j = 0$ and $\langle s_j \rangle_k = \langle s_j \rangle_k^{2 \cdot t} = 0$ for $j = 1, \ldots, \ell$ and $k = 1, \ldots, n$, and not return to step 1.*

   (h) If $\langle y_g \rangle_j \neq \langle y_g^* \rangle_j$ or $\langle y_g \rangle_j^{2 \cdot t} \neq \langle y_g^* \rangle_j^{2 \cdot t}$ then add $\{\mathcal{P}_i, \mathcal{P}_j\}$ to $Dispute$ for all $\mathcal{P}_j \in Agree_i$, and return to step 1 for all parallel executions.
   Note all parties will agree on the dispute having happened as they can all do the checks, which are on broadcast values.

8. Return $(\langle s_i \rangle, \langle s_i \rangle^{2 \cdot t})_{i=1}^{\ell}$.

Figure 61: Locally Produced, by $\mathcal{P}_i$, Checked Double Sharing of Multiple Values with Disputes. This is used for protocols in for the *nLarge* threshold profiles..

7.3.2.2 Randomness Extraction:

Given these two "batched" statistically secure VSS protocols we now present the randomness extraction methodology used to extract the "global" single and double sharings, see Figure 62 This randomness extraction is done using the Vandermonde matrices $M_{r,c}$ presented earlier in the standard manner.

Single and Double Sharing

The production of single and double sharings, of values $v_i \in R = GR(q, F)$, which are random and unknown to any party.

SingleSharing.Init(*Dispute*):
1. For $i \in [1, \ldots, \mathfrak{n}]$ do
   (a) Player $\mathcal{P}_i$ generates $s_j^{(i)} \in R$ for $j = 1, \ldots, \ell$.
   (b) $(\langle s_j^{(i)} \rangle)_{j=1}^{\ell} \leftarrow LocalSingleShare(\mathcal{P}_i, \{s_j^{(i)}\}_{j=1}^{\ell}, Dispute)$.
2. $cnt_0 \leftarrow 0, cnt_1 \leftarrow \mathfrak{n} - \mathfrak{t}$.

SingleSharing.Next(*Dispute*):
1. If $cnt_1 = \mathfrak{n} - \mathfrak{t}$ then
   (a) If $cnt_0 = \ell$ then call *SingleSharing.Init*(*Dispute*).
   (b) Locally compute
   $$(\langle v_j \rangle)_{j=1}^{\mathfrak{n}-\mathfrak{t}} \leftarrow (\langle s_{cnt_0}^{(i)} \rangle)_{i=1}^{\mathfrak{n}} \cdot M_{\mathfrak{n}, \mathfrak{n}-\mathfrak{t}}.$$

   (c) $cnt_0 \leftarrow cnt_0 + 1$.
   (d) $cnt_1 \leftarrow 0$.
2. $cnt_1 \leftarrow cnt_1 + 1$.
3. Return $\langle v_{cnt_1} \rangle$.

DoubleSharing.Init(*Dispute*):
1. For $i \in [1, \ldots, \mathfrak{n}]$ do
   (a) Player $\mathcal{P}_i$ generates $s_j^{(i)} \in R$ for $j = 1, \ldots, \ell$.
   (b) $(\langle s_j^{(i)} \rangle, \langle s_j^{(i)} \rangle^{2 \cdot \mathfrak{t}})_{j=1}^{\ell} \leftarrow LocalDoubleShare(\mathcal{P}_i, \{s_j^{(i)}\}_{j=1}^{\ell}, Dispute)$.
2. $cnt_0 \leftarrow 0, cnt_1 \leftarrow \mathfrak{n} - \mathfrak{t}$.

DoubleSharing.Next(*Dispute*):
1. If $cnt_1 = \mathfrak{n} - \mathfrak{t}$ then
   (a) If $cnt_0 = \ell$ then call *DoubleSharing.Init*(*Dispute*).
   (b) Locally compute
   $$(\langle v_j \rangle)_{j=1}^{\mathfrak{n}-\mathfrak{t}} \leftarrow (\langle s_{cnt_0}^{(i)} \rangle)_{i=1}^{\mathfrak{n}} \cdot M_{\mathfrak{n}, \mathfrak{n}-\mathfrak{t}}.$$
   $$(\langle v_j \rangle^{2 \cdot \mathfrak{t}})_{j=1}^{\mathfrak{n}-\mathfrak{t}} \leftarrow (\langle s_{cnt_0}^{(i)} \rangle^{2 \cdot \mathfrak{t}})_{i=1}^{\mathfrak{n}} \cdot M_{\mathfrak{n}, \mathfrak{n}-\mathfrak{t}}.$$

   (c) $cnt_0 \leftarrow cnt_0 + 1$.
   (d) $cnt_1 \leftarrow 0$.
2. $cnt_1 \leftarrow cnt_1 + 1$.
3. Return $(\langle v_{cnt_1} \rangle, \langle v_{cnt_1} \rangle^{2 \cdot \mathfrak{t}})$.

Figure 62: Interactive Single and Double Sharing. This is used for protocols in for the *nLarge* threshold profiles..

7.3.2.3 Offline Protocols: We can now present the main offline protocols for threshold profile *nLarge*, $MPC^L.Init(Dispute)$, $MPC^L.GenTriples(Dispute)$ and $MPC^O.NextRandom(Dispute)$ which we give in Figure 63. The error correction in line 1e of protocol $MPC^L.GenTriples$ works because for threshold profile *nLarge* we have assumed, following Design Decision 9, $\mathfrak{t} < \mathfrak{n}/4$, and not $\mathfrak{t} < \mathfrak{n}/3$. Extending the protocol, robustly, to the more relaxed setting of $\mathfrak{t} < \mathfrak{n}/3$ is possible using the techniques of [DN07]. However, the added complexity on the unhappy path creates a very complex protocol, which we feel is unlikely ever to be implemented.

MPC^L

Note, this assumes $\mathfrak{t} < \mathfrak{n}/4$, and synchronous networks.

$MPC^L.Init(Dispute)$:
1. Call $SingleSharing.Init(Dispute)$
2. Call $DoubleSharing.Init(Dispute)$
3. Define $\mathcal{T}, \mathcal{B} \leftarrow \emptyset$.

$MPC^L.GenTriples(Dispute)$:
1. Execute the following a "sufficient" number of times in parallel:
    (a) $\langle x \rangle \leftarrow SingleSharing.Next(Dispute)$.
    (b) $\langle y \rangle \leftarrow SingleSharing.Next(Dispute)$.
    (c) $(\langle v \rangle, \langle v \rangle^{2 \cdot \mathfrak{t}}) \leftarrow DoubleSharing.Next(Dispute)$.
    (d) Locally compute for all $\mathcal{P}_i$ the share $\langle d \rangle_i^{2 \cdot \mathfrak{t}} \leftarrow \langle x \rangle_i \cdot \langle y \rangle_i + \langle v \rangle^{2 \cdot \mathfrak{t}}$.
    (e) $d \leftarrow RobustOpen(\mathcal{P}, \langle d \rangle^{2 \cdot \mathfrak{t}})$.
    (f) $\langle z \rangle \leftarrow d - \langle v \rangle$.
    (g) $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\langle x \rangle, \langle y \rangle, \langle z \rangle)\}$.

$MPC^L.NextRandom(Dispute)$:
1. $\langle x \rangle \leftarrow SingleSharing.Next(Dispute)$.
2. Return $\langle x \rangle$.

Figure 63: The Offline Procedures for Threshold Profiles in Category *nLarge*.

7.4 Layer Three

We summarize the relationship between the protocols in Layer Three in Figure 64–Figure 66.
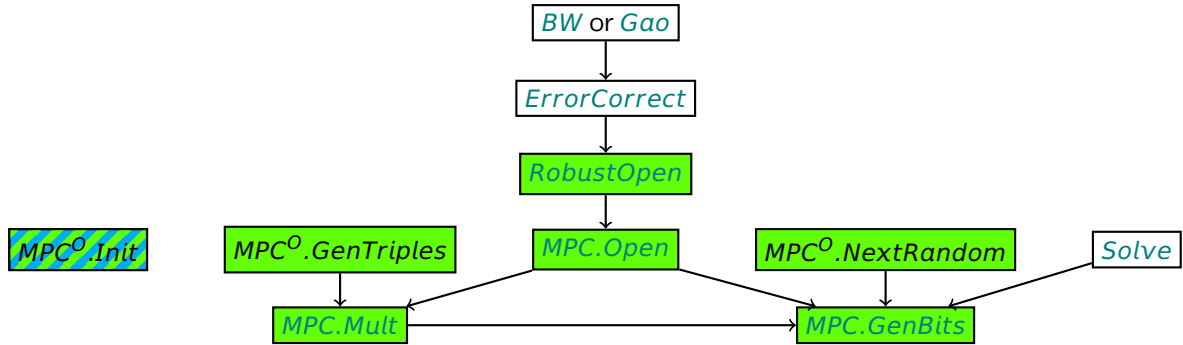
Figure 64: Online MPC Protocols for All Threshold Profile Categories and $q = 2^k$. Those protocols marked MPC^O come from the offline phase; thus depending on the profile one replace O with S or L. The protocols in the online phase can work in **asynchronous** networks, even though the offline phase assumes **synchronous** networks. Depending on the implementation choices, one could consider *MPC.GenBits* as actually part of the Offline phase.
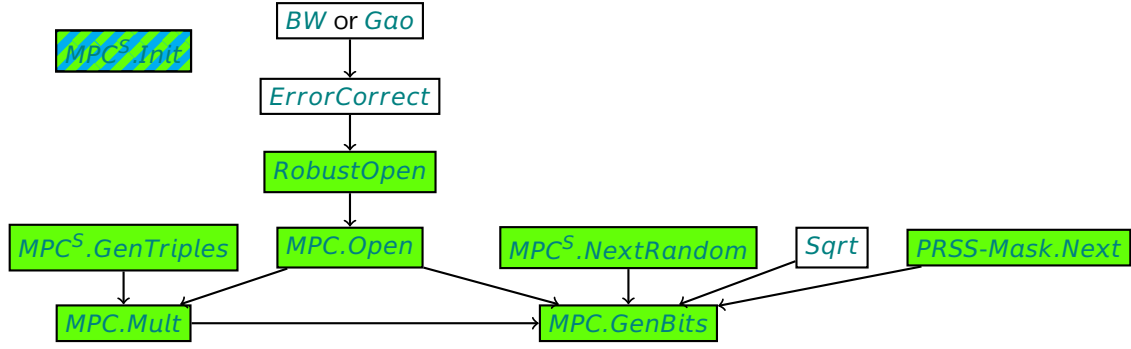
Figure 65: Online MPC Protocols for *nSmall* Threshold Profile Categories when $q = p_1 \ldots p_t$. Those protocols marked MPC^S come from the offline phase. The protocols in the online phase can work in **asynchronous** networks, even though the offline phase assumes **synchronous** networks. Depending on the implementation choices, one could consider *MPC.GenBits* as actually part of the Offline phase.
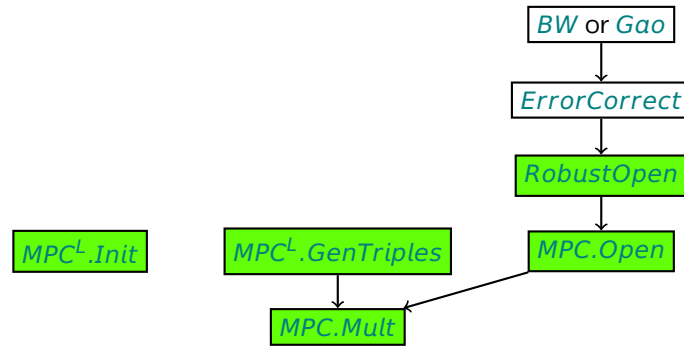
Figure 66: Online MPC Protocols for *nLarge* Threshold Profile Category when $q = p_1 \cdots p_k$. Those protocols marked MPC^L come from the offline phase. The protocols in the online phase can work in **asynchronous** networks, even though the offline phase assumes **synchronous** networks. There is no *MPC.GenBits* protocol here, thus this set of protocols are just for expository purposes. They are not used in any of our threshold protocols.

7.4.1 Online MPC Protocol for All Threshold Profile Categories

Here we describe the online phase of the MPC algorithm. Our MPC online algorithm will be used in a rather special situation; in particular there is no need for an input routine (recall Design Decision 6). The online algorithms are all robustly secure, over asynchronous networks, assuming $t < n/3$. However, the underlying offline algorithms may not meet these requirement. In particular none of our offline methods are valid over asynchronous networks, and we can only obtain $t < n/3$ for the offline phase for threshold profile categories *nSmall*, otherwise we need to assume $t < n/4$.

An algorithm built on top of the MPC engine can perform linear operations (for free), request random pre–computed bits, perform multiplication operations and output a shared value to all parties. These routines are presented in Figure 67.

7.4.1.1 Multiplication: Our multiplication routine is the basic Beaver multiplication algorithm. The two *MPC.Open*'s in the multiplication routine can be run in parallel.

7.4.1.2 *MPC.Mask*: One can think of our operation *PRSS-Mask.Next* as either a subroutine at Layer Two, or as a special command of our MPC protocol, which one could denote *MPC.Mask*. For ease of exposition of the protocols we take the former approach, by calling it *PRSS-Mask.Next*. However, when presenting the security rationale behind this operation, in Section 8.4.1.2, we think of it as an extension to the online MPC functionality.

MPC Online

We utilize the notation MPC^O to denote a call to the offline process, where $O = S$ or $O = L$ depending on the specific parameters for our MPC instantiation.

$MPC.Open(\langle x \rangle)$:
     1. Execute $RobustOpen(\{\mathcal{P}_1, \ldots, \mathcal{P}_\mathfrak{n}\}, \langle x \rangle)$.

$MPC.Mult(\langle x \rangle, \langle y \rangle)$:
     1. If $\mathcal{T} = \varnothing$ then execute $MPC^O.GenTriples(Dispute)$.
     2. Write $\mathcal{T} = \{(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)\}_{i=1}^{N}$.
     3. $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(\langle a_1 \rangle, \langle b_1 \rangle, \langle c_1 \rangle)\}$.
     4. $\langle \epsilon \rangle \leftarrow \langle x \rangle + \langle a_1 \rangle$.
     5. $\langle \rho \rangle \leftarrow \langle y \rangle + \langle b_1 \rangle$.
     6. $\epsilon \leftarrow MPC.Open(\langle \epsilon \rangle)$.
     7. $\rho \leftarrow MPC.Open(\langle \rho \rangle)$.
     8. $\langle z \rangle \leftarrow \langle c_1 \rangle + \epsilon \cdot \langle y \rangle - \rho \cdot \langle a \rangle$.

$MPC.GenBits(v)$: This algorithm returns v bits for use in further algorithms
     1. While $|\mathcal{B}| < v$ then execute the following a "sufficient" (or "desired") number of times in parallel
        (a) $\langle a \rangle \leftarrow MPC^O.NextRandom(Dispute)$.
        (b) $\langle s \rangle \leftarrow MPC.Mult(\langle a \rangle, \langle a \rangle)$.
        (c) If q odd then
           i. If profile $nLarge$ then $abort$.
           ii. Let p_L denote the largest prime divisor of q.
           iii. $s \leftarrow MPC.Open(\langle s \rangle)$.
           iv. If $s \equiv 0 \mod p_L$ then return to step 1a.
           v. $c \leftarrow Sqrt(s, p_L)$.
           vi. $\langle v \rangle \leftarrow \langle a \rangle / c$.
           vii. $\langle b \rangle \leftarrow (1 + \langle v \rangle) / 2$.
           viii. $\langle r \rangle \leftarrow PRSS\text{-}Mask.Next(1, stat) + 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat}$.
           ix. $c \leftarrow MPC.Open(\langle b + r \rangle)$.
           x. $t \leftarrow c \pmod{p_L}$.
           xi. $\langle b \rangle \leftarrow t - \langle r \rangle$.
        (d) Else
           i. $\langle v \rangle \leftarrow \langle a \rangle + \langle s \rangle$.
           ii. $v \leftarrow MPC.Open(\langle v \rangle)$.
           iii. $r \leftarrow Solve(v)$. Using the method from Section 7.1.5.1.
           iv. $d \leftarrow (-1 - 2 \cdot r) \pmod{q}$.
           v. $\langle b \rangle \leftarrow (\langle a \rangle - r) / d$.
        (e) $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle b \rangle\}$.
     2. Write $\mathcal{B} = \{\langle b_i \rangle\}_{i=1}^{N}$.
     3. $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\langle b_i \rangle\}_{i=1}^{v}$.
     4. Return $\{\langle b_1 \rangle, \ldots, \langle b_v \rangle\}$.

Figure 67: Online MPC Routines.

7.4.1.3 MPC Algorithms: On top of this MPC engine (consisting of addition, multiplication, opening and masking of secret shared values) we can run algorithms. As long as these algorithms only make use of these basic operations the resulting algorithms will only leak the data which is leaked by their outputs. After all this is what an MPC algorithm is designed to do. To aid exposition however, we treat the bit generation algorithms as part of the MPC engine itself (i.e. we place them at Layer Three, rather than perhaps at a Layer Three–point–Five). Thus the two methods for bit generation in Figure 67 can be seen as simple MPC programs more akin to the MPC programs we will execute at Layer Four.

7.4.1.4 Bit Generation when $q = 2^k$:   The bit–generation method, when $q = 2^k$, is a direct translation of the "in–the–clear" method from Section 7.1.5 to the MPC environment.

7.4.1.5 Bit Generation when $q = p_1 \cdots p_k$:   When $q = p_1 \cdots p_k$ we noted in Section 7.1.5 that the method is more complex when $k > 1$. The "in–the–clear" method produces can produce a shared bit modulo any single prime divisor of q, but obtaining a consistent shared bit is the problem. Following Design Decision 13 we present a solution to this problem only in the case of threshold profiles *nSmall*. Our solution method makes usage of the *PRSS* in a key way; in particular the *PRSS-Mask.Next* given earlier is shifted to ensure we always work with positive numbers, avoiding any mismatch in the mapping of negative numbers mod p_L and mod q. A proof of this method is given in Theorem 7 later. The method requires that we have

$$p_L > nSmallBnd \cdot 2^{stat+3} > 8 \cdot \binom{n}{t} \cdot 2^{stat},$$

which holds by Parameter Choice 2.

A solution to producing shared random bits in threshold profile *nLarge* is possible, using an adaption of the checking techniques of [RST$^+$22, RW19, EGK$^+$20]. However, these methods are relatively complex, and when applied out-of-the-box they only result in protocols which are active–with–abort secure, and not robustly secure. Thus these methods are worthy of more research in the context of the Shamir Sharings over Galois rings discussed in this document.

7.5 Layer Four

We discuss the three schemes algorithms (BGV, BFV and TFHE) in turn. In each scheme we need to see how to map the scheme parameters *Q, P, N* etc onto the parameters of the underlying MPC/secret sharing scheme q, $\mathfrak{n}$, $\mathfrak{t}$ etc. And how these are related, and what constrains are placed upon the various parameters. We summarize the relationship between the protocols in Layer Four in Figure 68– Figure 74.
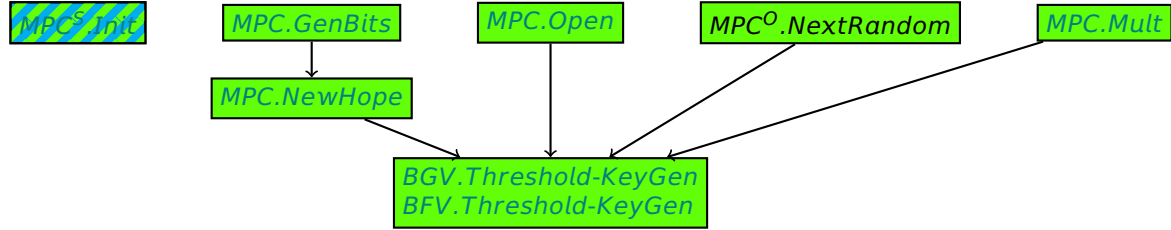
Figure 68: Threshold Key Generation for Threshold Profile Category *nSmall* (BGV/BFV). These protocols can work in **asynchronous** networks.
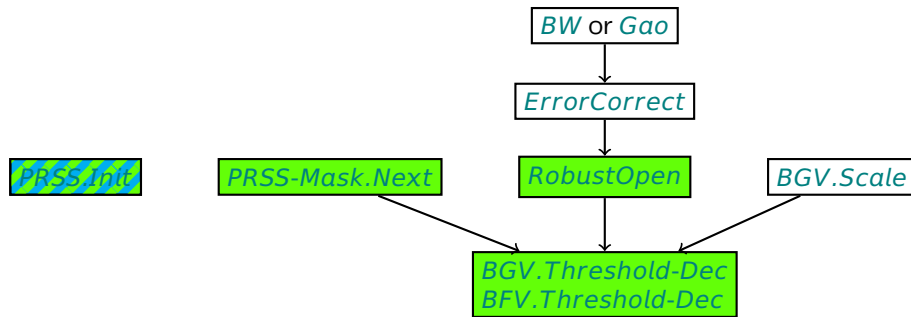
Figure 69: Threshold Decryption for Threshold Profile Category *nSmall* (BGV/BFV). These protocols can work in **asynchronous** networks.
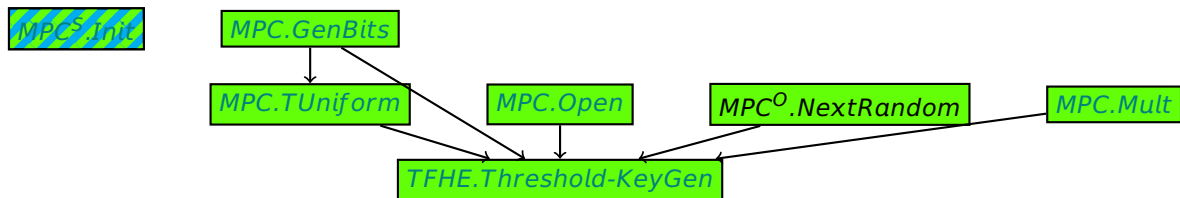
Figure 70: Threshold Key Generation for all Threshold Profile Categories (TFHE). These protocols can work in **asynchronous** networks.
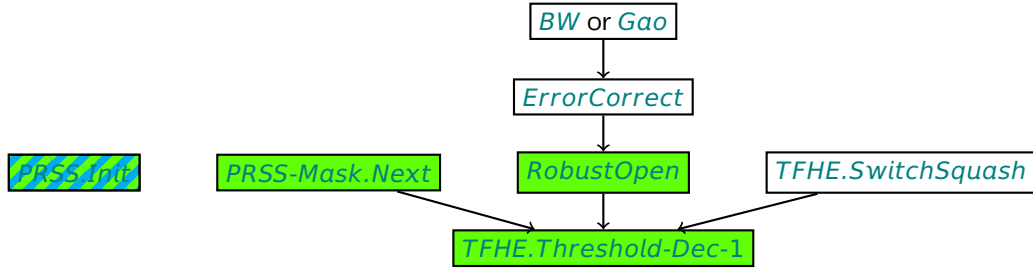
Figure 71: Threshold Decryption (Version 1) for Threshold Profile Category *nSmall* (TFHE). These protocols can work in **asynchronous** networks.
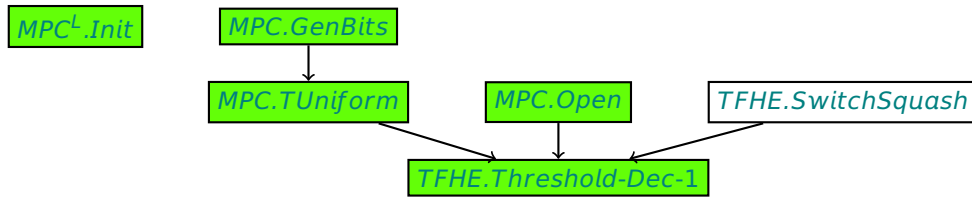
Figure 72: Threshold Decryption (Version 1) for Threshold Profile Category *nLarge* (TFHE). These protocols can work in **asynchronous** networks.
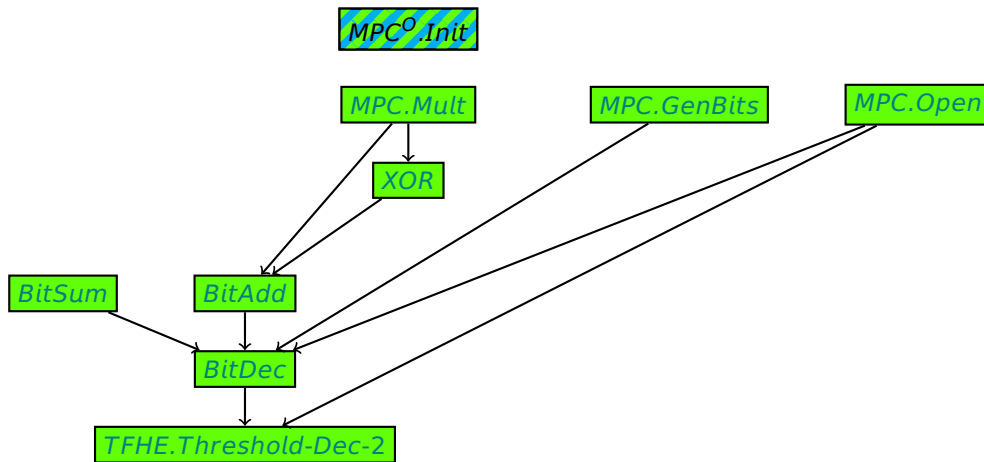
Figure 73: Threshold Decryption (Version 2) for All Threshold Profile Categories (TFHE). These protocols can work in **asynchronous** networks.
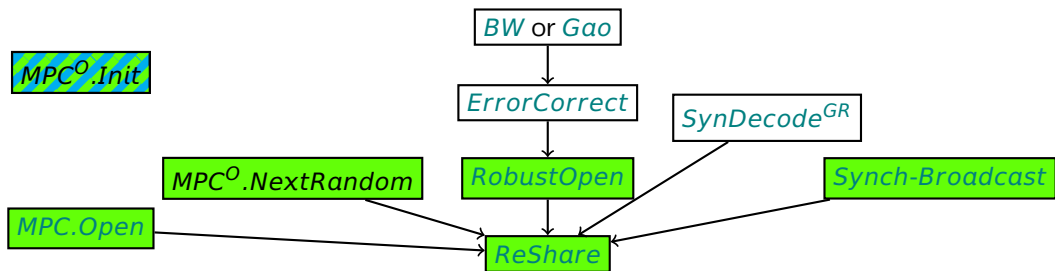
Figure 74: Resharing Protocol for all Threshold Profiles. These protocols work in **synchronous** networks.

7.5.1 Distributions

We first discuss how to generate our random distributions in our MPC engines, i.e. we want to sample from $NewHope(N, B)$ and $TUniform(N, -2^b, 2^b)$ within the secret shared domain. These algorithms are immediate, given our ability to generate shared random bits, and are given in Figure 75.

Secret Shared Distributions

$MPC.NewHope(N, B)$:
    1. $\langle \mathbf{b} \rangle = (\langle b_1 \rangle, \ldots, \langle b_{2 \cdot N \cdot B} \rangle) \leftarrow MPC.GenBits(2 \cdot N \cdot B)$.
    2. $\langle \mathbf{e} \rangle = (\langle e_1 \rangle, \ldots, \langle e_N \rangle) \leftarrow (\langle 0 \rangle, \ldots, \langle 0 \rangle)$.
    3. For $i \in [1, \ldots, N]$ do
        (a) For $j \in [1, \ldots, B]$ do
   i. $t \leftarrow 2 \cdot (i - 1) \cdot B + 2 \cdot (j - 1) + 1$.
            ii. $\langle e_i \rangle \leftarrow \langle e_i \rangle + \langle b_t \rangle - \langle b_{t+1} \rangle$.
    4. Return $\langle \mathbf{e} \rangle$.

$MPC.TUniform(N, -2^b, 2^b)$:
    1. $\langle \mathbf{b} \rangle = (\langle b_1 \rangle, \ldots, \langle b_{N \cdot (b+2)} \rangle) \leftarrow MPC.GenBits(N \cdot (b + 2))$.
    2. For $i \in [1, \ldots, N]$ do
        (a) $r \leftarrow (i - 1) \cdot (b + 2)$.
        (b) $\langle e_i \rangle \leftarrow \langle b_{r+b+2} \rangle - 2^b$.
        (c) For $j \in [1, \ldots, b + 1]$ do
            i. $\langle e_i \rangle \leftarrow \langle e_i \rangle + \langle b_{r+j} \rangle \cdot 2^{j-1}$.
    3. Return $\langle \mathbf{e} \rangle$.

Figure 75: Protocols to Secret Shared Samples from $NewHope(N, B)$ and $TUniform(N, -2^b, 2^b)$.

7.5.2 BGV

Recall we only provide threshold BGV protocols in threshold profile $nSmall$. For threshold key generation for BGV we will utilize our MPC/secret sharing schemes with $q = Q \cdot R$, however when we switch to threshold decryption operations we will switch to using $q = Q_1$. This switching is simple, since $Q_1 | Q \cdot R$, as given a valid secret sharing modulo $Q \cdot R$ we can obtain a valid secret sharing modulo Q_1 simply by each player locally reducing their share modulo Q_1.

7.5.2.1 Threshold Key Generation: Recall we have selected $Q = Q_1 \cdots Q_L$ such that, by Parameter Choice 5,

$$Q_1 \approx 4 \cdot nSmallBnd \cdot 2^{stat} \cdot E_M \cdot B_{Mult}.$$

In particular this means (for key generation) that the largest prime p_L dividing Q is larger than $nSmallBnd \cdot 2^{stat+3}$. Hence, Parameter Choice 2 is automatically satisfied, and our bit generation technique will work for our secret sharing modulo $q = Q \cdot R$. This means that our threshold key generation algorithm for BGV is immediate, and we produce BGV keys with exactly the same noise distribution as we would in the clear.

The algorithm is simply a thresholdization (using our generic MPC machinery) of $KeyGen$ from Figure 8, which we present in Figure 76. Notice that all the arithmetic operations performed in $BGV.Threshold\text{-}KeyGen$ on shared values are all linear; except for the generation of the sharing of $\mathfrak{sk}^2$. Thus, apart from the generation of the random bits and the opening of the final sharings, the entire algorithm is completely local, except for the layer of multiplications (in line 10) needed to compute $\mathfrak{sk}^2$. This multiplication can be implemented either using a naive loop, or using an NTT/iNTT methodology as described in Section 5.3. In either way it requires a single parallel execution of $O(N^2)$ (resp. N when using an NTT based

---

BGV Threshold Key Generation

BGV.Threshold-KeyGen(N, Q, P, B, R):

Unless otherwise marked, all the secret sharings in this algorithm are modulo $q = T = Q \cdot R$.

1. $\langle \mathfrak{sk} \rangle \leftarrow$ *MPC.NewHope*($N, 1$).
2. $\langle \mathfrak{pk}_a \rangle \leftarrow MPC^O$.*NextRandom*().
3. $\langle \mathfrak{pk}'_a \rangle \leftarrow MPC^O$.*NextRandom*().
4. $\mathfrak{pk}_a \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}_a \rangle$).
5. $\mathfrak{pk}'_a \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}'_a \rangle$).
6. $\mathfrak{pk}_a \leftarrow \mathfrak{pk}_a \pmod{Q}$.
7. $\langle \mathbf{e}_{\mathfrak{pk}} \rangle \leftarrow$ *MPC.NewHope*(N, B).
8. $\langle \mathfrak{pk}_b \rangle \leftarrow \mathfrak{pk}_a \odot \langle \mathfrak{sk} \rangle + P \cdot \langle \mathbf{e}_{\mathfrak{pk}} \rangle$.
9. $\langle \mathbf{e}'_{\mathfrak{pk}} \rangle \leftarrow$ *MPC.NewHope*(N, B).
10. $\langle \mathbf{s} \rangle \leftarrow \langle \mathfrak{sk} \rangle \odot \langle \mathfrak{sk} \rangle$.
11. $\langle \mathfrak{pk}'_b \rangle \leftarrow \mathfrak{pk}'_a \odot \langle \mathfrak{sk} \rangle + P \cdot \langle \mathbf{e}'_{\mathfrak{pk}} \rangle - R \cdot \langle \mathbf{s} \rangle$.
12. $\mathfrak{pk}_b \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}_b \rangle$).
13. $\mathfrak{pk}'_b \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}'_b \rangle$).
14. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_b \pmod{Q}$.
15. $\langle \mathfrak{sk} \rangle_{Q_1} \leftarrow \langle \mathfrak{sk} \rangle \pmod{Q_1}$, i.e. restrict $\langle \mathfrak{sk} \rangle$ to a secret sharing modulo Q_1.
16. $\mathfrak{pk} \leftarrow \{(\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}'_a, \mathfrak{pk}'_b)\}$.
17. Return $(\mathfrak{pk}, \langle \mathfrak{sk} \rangle_{Q_1})$.

Figure 76: BGV Threshold Key Generation, for Threshold Profile *nSmall*.

methodology[18]) MPC multiplications. Note, NTT based multiplication can also be used in lines 8 and 11 if desired.

Note, the openings on lines 4 and 5 can be performed in parallel, as can the openings on lines 12 and 13.

7.5.2.2   Threshold Decryption: Recall at this point we hold a secret key in secret shared form $\langle \mathfrak{sk} \rangle$, where the sharing is performed modulo $q = Q_1$. Also recall, that a BGV ciphertext is a tuple $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B)$ where we have

$$Q^{(\ell)} = Q_1 \cdots Q_\ell,$$

$$\mathbf{c}_0, \mathbf{c}_1 \in \mathbf{R}_{Q^{(\ell)}},$$

$$\mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk} = \mathbf{m} + P \cdot \mathbf{e} \pmod{Q^{(\ell)}}$$

$$\|\mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk}\|^{can} \leq B.$$

To proceed with threshold decryption we transform the ciphertext first from level ℓ to level one. Assuming the (standard BGV) parameters have been set up correctly, this will result in a ciphertext of the form $\mathfrak{ct}' = (\mathbf{c}'_0, \mathbf{c}'_1, 1, B')$ where we have

$$B' < B_{Mult},$$

and hence we will have the infinity norm bound

$$\|\mathbf{c}_0 - \mathbf{c}_1 \odot \mathfrak{sk}\|_\infty \leq E_M \cdot B_{Mult}.$$

---

[18]Applying the NTT/iNTT is a linear, and hence local, operation.

In setting the parameters for the threshold version for BGV, recall, we selected Q_1 so that

$$2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \cdot E_M \cdot B_{Mult} < Q_1/2.$$

Indeed we selected

$$Q_1 \approx 4 \cdot nSmallBnd \cdot 2^{stat} \cdot E_M \cdot B_{Mult}.$$

This means there is enough "noise gap" between the infinity norm of the pre–decryption of $\mathfrak{ct}'$ and the modulus Q_1 in order to apply noise–flooding to the pre–decryption. The noise–flooding can then be applied, as we are in threshold profile *nSmall*, using the *PRSS-Mask* operations, as described in Figure 77.

BGV Threshold Decryption

BGV.Threshold-Dec($\mathfrak{ct}$, $\langle \mathfrak{sk} \rangle$):
1. $\mathfrak{ct}' = (\mathbf{c}'_0, \mathbf{c}'_1, 1, B') \leftarrow$ *BGV.Scale*($\mathfrak{ct}$, 1).
2. $\langle \mathbf{p} \rangle \leftarrow \mathbf{c}'_0 - \langle \mathfrak{sk} \rangle \odot \mathbf{c}'_1$.
3. $\langle \mathbf{t} \rangle \leftarrow$ *PRSS-Mask.Next*($E_M \cdot B_{Mult}/P$, *stat*); note this is in fact N calls to *PRSS-Mask.Next* as $\mathbf{t}$ is a vector of length N.
4. $\langle \mathbf{c} \rangle \leftarrow \langle \mathbf{p} \rangle + P \cdot \langle \mathbf{t} \rangle$.
5. $\mathbf{c} \leftarrow$ *RobustOpen*($\{ \mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}} \}$, $\langle \mathbf{c} \rangle$); again this in N parallel calls.
6. $\mathbf{m} \leftarrow \mathbf{c} \pmod{P}$.
7. Return $\mathbf{m}$.

Figure 77: BGV Threshold Decryption, for Threshold Profile *nSmall*.

Note in *BGV.Threshold-Dec* the N parallel calls to *PRSS-Mask.Next* and *RobustOpen* can be reduced if it is known before hand that the number of coefficients returned in the message is actually less than N.

This distributed decryption protocol is secure, since the opening of the value $\mathbf{c}$ leaks no information about the noise value inside $\mathfrak{ct}'$, since the addition of the value returned by the *PRSS-Mask.Next* function "floods" the value. This follows from Theorem 6; but we will provide a full simulation proof later.

For correctness note that the value $\mathbf{c}$ has infinity norm bounded by (with overwhelming probability)

$$E_M \cdot B_{Mult} \cdot \left(1 + 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \right) \approx 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \cdot E_M \cdot B_{Mult}, < Q_1/2.$$

Thus there is no wrap–around error introduced by adding on the value $P \cdot \mathbf{t}$. That we obtain the same message then follows because we added on a multiple of P; which does not affect the final decrypted value.

7.5.3 BFV

Given our earlier discussion that BFV is just a "converted" version of BGV, the translation of the in–the–clear BFV key generation and decryption operations to their threshold versions is then immediate.

7.5.3.1 Threshold Key Generation: BFV threshold key generation follows from Figure 12, and is given in Figure 78 Notice that we only provide a solution for threshold profile *nSmall*, for exactly the same reason as for threshold BGV, namely that the generation of shared random bits for composite odd q is beyond the scope of this document. The only change from the BGV threshold key generation method from Figure 76 is in line 8

7.5.3.2 Threshold Decryption: Recall for a BFV ciphertext $\mathfrak{ct} = (\mathbf{c}_0, \mathbf{c}_1, \ell, B_{\mathfrak{ct}})$ the bound $B_{\mathfrak{ct}}$ is on the canonical norm of the pre–decryption of the ciphertext in BGV format. Also recall a *Scale* operation

---

BFV Threshold Key Generation

BFV.Threshold-KeyGen(N, Q, P, B, R):
  Unless otherwise marked, all the secret sharings in this algorithm are modulo $q = T = Q \cdot R$.
  1. $\langle \mathfrak{sk} \rangle \leftarrow$ *MPC.NewHope*$(N, 1)$.
  2. $\langle \mathfrak{pk}_a \rangle \leftarrow MPC^O.NextRandom()$.
  3. $\langle \mathfrak{pk}'_a \rangle \leftarrow MPC^O.NextRandom()$.
  4. $\mathfrak{pk}_a \leftarrow$ *MPC.Open*$(\langle \mathfrak{pk}_a \rangle)$.
  5. $\mathfrak{pk}'_a \leftarrow$ *MPC.Open*$(\langle \mathfrak{pk}'_a \rangle)$.
  6. $\mathfrak{pk}_a \leftarrow \mathfrak{pk}_a \pmod{Q}$.
  7. $\langle \mathbf{e}_{\mathfrak{pk}} \rangle \leftarrow$ *MPC.NewHope*(N, B).
  8. $\langle \mathfrak{pk}_b \rangle \leftarrow \mathfrak{pk}_a \odot \langle \mathfrak{sk} \rangle + \langle e_{\mathfrak{pk}} \rangle$.
  9. $\langle \mathbf{e}'_{\mathfrak{pk}} \rangle \leftarrow$ *MPC.NewHope*(N, B).
  10. $\langle \mathbf{s} \rangle \leftarrow \langle \mathfrak{sk} \rangle \odot \langle \mathfrak{sk} \rangle$.
  11. $\langle \mathfrak{pk}'_b \rangle \leftarrow \mathfrak{pk}'_a \odot \langle \mathfrak{sk} \rangle + P \cdot \langle \mathbf{e}'_{\mathfrak{pk}} \rangle - R \cdot \langle \mathbf{s} \rangle$.
  12. $\mathfrak{pk}_b \leftarrow$ *MPC.Open*$(\langle \mathfrak{pk}_b \rangle)$.
  13. $\mathfrak{pk}'_b \leftarrow$ *MPC.Open*$(\langle \mathfrak{pk}'_b \rangle)$.
  14. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_b \pmod{Q}$.
  15. $\langle \mathfrak{sk} \rangle_{Q_1} \leftarrow \langle \mathfrak{sk} \rangle \pmod{Q_1}$, i.e. restrict $\langle \mathfrak{sk} \rangle$ to a secret sharing modulo Q_1.
  16. $\mathfrak{pk} \leftarrow \{ (\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}'_a, \mathfrak{pk}'_b) \}$.
  17. Return $(\mathfrak{pk}, \langle \mathfrak{sk} \rangle_{Q_1})$.

Figure 78: BFV Threshold Key Generation, for Threshold Profile *nSmall*.

down to level one needs to be applied to perform threshold decryption, and this is only defined for BGV format ciphertexts. Thus, if a ciphertext is presented in BFV format, we actually simply convert it to BGV format and then apply the BGV threshold decryption operation. Thus threshold BFV is defined as in Figure 79.

BFV Threshold Decryption

BFV.Threshold-Dec$(\mathfrak{ct}, \langle \mathfrak{sk} \rangle)$:
  1. $\mathfrak{ct}' = BFV.toBGV(\mathfrak{ct})$.
  2. $\mathbf{m} \leftarrow BGV.Threshold\text{-}Dec(\mathfrak{ct}', \langle \mathfrak{sk} \rangle)$.
  3. Return $\mathbf{m}$.

Figure 79: BFV Threshold Decryption, for Threshold Profile *nSmall*.

7.5.4   TFHE

For TFHE we provide two different methodologies for threshold decryption. The first is a one round variant, which requires us to boost the ciphertext modulus from the typical $Q = 2^{64}$ to one of $\overline{Q} = 2^{128}$. Thus for this variant we need to execute our secret sharing operations (and hence key generation below) using the modulus $\overline{Q}$. The second variant is a higher round complexity variant, which means we can maintain the ciphertext modulus of $Q = 2^{64}$. In this second case we run the threshold key generation using secret sharing modulo Q.

7.5.4.1   Threshold Key Generation: As remarked above this method is either executed using secret sharing modulo $q = \overline{Q} = 2^{128}$ or $q = Q = 2^{64}$. The algorithm is simply a thresholdization (using our generic MPC machinery) of *KeyGen* from Figure 17, which we present in Figure 81. This in turn uses MPC

versions of the encryption algorithms from Figure 14, which we present in Figure 80. The method is the same whether one is in threshold profile *nSmall* or threshold profile *nLarge* (the only thing changing being the underlying MPC mechanics). An astute reader may notice that, contrary to *KeyGen* from Figure 17, the thresholdized version in Figure 81 always outputs shares of the *LWE* secret key as the first component of $\mathfrak{sk}$ irrespective of the *type*. This is because, for efficiency reasons, our second distributed decryption protocol described in Figure 85 first performs a key switch if *type = F-GLWE*.

The MPC Internal TFHE Encryption Operations

$MPC.Enc^{LWE}(\langle m \rangle, \langle \mathbf{s} \rangle; XOF, P', Q, \ell, flag)$:
  1. $\mathbf{a} \leftarrow Expand^{LWE}(XOF, Q, \ell)$.
  2. $\langle e \rangle \leftarrow MPC.TUniform(1, -2^{b_\ell}, 2^{b_\ell})$.
  3. $\langle b \rangle \leftarrow \mathbf{a} \cdot \langle \mathbf{s} \rangle + \langle e \rangle + (Q/P') \cdot \langle m \rangle$.
  4. If $flag$ return $(\mathbf{a}, \langle b \rangle)$, else return $\langle b \rangle$.

$MPC.Enc^{GLWE}(\langle \mathbf{m} \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, P', Q, N, w, flag)$:
  1. $(\mathbf{a}_0, \ldots, \mathbf{a}_{w-1}) \leftarrow Expand^{GLWE}(XOF, Q, N, w)$.
  2. $\langle \mathbf{e} \rangle \leftarrow MPC.TUniform(N, -2^{b_{w \cdot N}}, 2^{b_{w \cdot N}})$.
  3. $\langle \mathbf{b} \rangle \leftarrow \sum_{i=0}^{w-1} \mathbf{a}_i \odot \langle \mathbf{s}_i \rangle + \langle \mathbf{e} \rangle + (Q/P') \cdot \langle \mathbf{m} \rangle \pmod{Q}$.
  4. If $flag$ return $(\mathbf{a}_0, \ldots, \mathbf{a}_{w-1}, \langle \mathbf{b} \rangle)$, else return $\langle \mathbf{b} \rangle$.

$MPC.Enc^{Lev}(\langle m \rangle, \langle \mathbf{s} \rangle; XOF, \beta, Q, \ell, \nu, flag)$:
  1. For $i \in [0, \ldots, \nu - 1]$ do
     (a) $\langle \mathfrak{ct}_i \rangle \leftarrow MPC.Enc^{LWE}(m, \mathbf{s}; XOF, \beta^{i+1}, Q, \ell, flag)$.
         We abuse notation here as not all of $\mathfrak{ct}_i$ may be secret shared.
  2. Return $(\langle \mathfrak{ct}_0 \rangle, \ldots, \langle \mathfrak{ct}_{\nu-1} \rangle)$.

$MPC.Enc^{GLev}(\langle \mathbf{m} \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, \beta, Q, N, w, \nu, flag)$:
  1. For $i \in [0, \ldots, \nu - 1]$ do
     (a) $\langle \mathfrak{ct}_i \rangle \leftarrow MPC.Enc^{GLWE}(\langle \mathbf{m} \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, \beta^{i+1}, Q, N, w, flag)$.
         We abuse notation here as not all of $\mathfrak{ct}_i$ may be secret shared.
  2. Return $(\langle \mathfrak{ct}_0 \rangle, \ldots, \langle \mathfrak{ct}_{\nu-1} \rangle)$.

$MPC.Enc^{GGSW}(\langle m \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, Q, N, w, \beta, \nu, flag)$:
  1. For $i \in [0, \ldots, w - 1]$ do
     (a) $\langle \mathbf{t}_i \rangle \leftarrow -MPC.Mult(\langle \mathbf{s}_i \rangle, \langle m \rangle)$.
         Note, this requires interaction.
     (b) $\langle \mathfrak{ct}_i \rangle \leftarrow MPC.Enc^{GLev}(\langle \mathbf{t}_i \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, \beta, Q, N, w, \nu, flag)$.
         Again, we abuse notation here as not all of $\mathfrak{ct}_i$ maybe secret shared.
  2. $\langle \mathfrak{ct}_w \rangle \leftarrow MPC.Enc^{GLev}(\langle m \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, \beta, Q, N, w, \nu, flag)$.
  3. Return $(\langle \mathfrak{ct}_0 \rangle, \ldots, \langle \mathfrak{ct}_w \rangle)$.

Figure 80: The MPC Internal TFHE Encryption Operations.

Note that all the methods in Figure 80 on secret sharings, are either generation of random bits or purely linear operations; except for line 1a of the method for $MPC.Enc^{GGSW}$, which requires a secure multiplication algorithm to be performed. Thus after an "offline" phase used to generate the bits, the operations in Figure 80 are purely local operations, except for this single execution of multiplications. However, clearly all of these multiplications can be done in parallel. Hence, the total round cost of all the calls to the functions in Figure 80 during the entire threshold key generation algorithm will be one; except for the rounds needed to generate the shared random bits. Indeed, examining Figure 81 we see that these are the only multiplications required during key generation; and thus (except for the offline

TFHE.Threshold-KeyGen$(P, Q, type, \hat{\ell}, \ell, N, w, \beta_{pksk}, \beta_{ksk}, \beta_{bk}, \nu_{pksk}, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \overline{Q}, \overline{\beta_{bk}},$
$\overline{\nu_{bk}}, flag, \overline{flag})$: Unless otherwise marked, all the secret sharings in this algorithm are modulo $q = Q$ or $\overline{Q}$, depending on the method chosen for threshold decryption.

1. If $\overline{flag}$ then $\langle \cdot \rangle$ denotes secret sharing modulo $q = \overline{Q}$, otherwise it denotes secret sharing modulo $q = Q$.
2. For $i = 1, \ldots, \lceil sec/(d \cdot \log_2 q) \rceil$ do
   (a) $\langle seed_i \rangle \leftarrow MPC^O.NextRandom()$.
   (b) $seed_i \leftarrow MPC.Open(\langle seed_i \rangle)$.
3. $seed \leftarrow (seed_1 \| \ldots \| seed_{\lceil sec/\log_2 q \rceil}) \pmod{2^{sec}}$.
4. *XOF.Init*$(seed, DSep(TFHE))$.
5. $\langle \hat{\mathbf{s}} \rangle \leftarrow MPC.GenBits(\hat{\ell})$.
6. $\langle \mathbf{s} \rangle \leftarrow MPC.GenBits(\ell)$.
7. For $i \in [0, \ldots, w-1]$ do $\langle \mathbf{s}_i \rangle \leftarrow MPC.GenBits(N)$.
8. $\mathfrak{pk}_a \leftarrow XOF.Next(\hat{\ell}, Q)$.
9. $\langle \mathbf{e} \rangle \leftarrow MPC.TUniform(\hat{\ell}, -2^{b_i}, 2^{b_i})$.
10. $\langle \mathfrak{pk}_b \rangle \leftarrow \mathfrak{pk}_a \odot \overleftrightarrow{\langle \hat{\mathbf{s}} \rangle} + \langle \mathbf{e} \rangle$.
    Note this is a linear operation.
11. $\mathfrak{pk}_b \leftarrow MPC.Open(\langle \mathfrak{pk}_b \rangle) \pmod{Q}$.
12. For $i \in [0, \ldots, w-1]$ do
    (a) For $j \in [0, \ldots, N-1]$ do
        i. $\langle KSK_{i,j} \rangle \leftarrow MPC.Enc^{Lev}(\langle \mathbf{s}_i[j] \rangle, \langle \mathbf{s} \rangle; XOF, \beta_{ksk}, Q, \ell, \nu_{ksk}, flag)$.
        ii. $KSK_{i,j} \leftarrow MPC.Open(\langle KSK_{i,j} \rangle) \pmod{Q}$.
13. If $\overline{flag}$ then
    (a) For $i \in [0, \ldots, \overline{w}-1]$ do $\langle \overline{\mathbf{s}}_i \rangle \leftarrow MPC.GenBits(\overline{N})$.
14. For $i \in [0, \ldots, \ell-1]$ do
    (a) $\langle BK_i \rangle \leftarrow MPC.Enc^{GGSW}(\langle \mathbf{s}[i] \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, Q, N, w, \beta_{bk}, \nu_{bk}, flag)$.
    (b) $BK_i \leftarrow MPC.Open(\langle BK_i \rangle) \pmod{Q}$.
    (c) If $\overline{flag}$ then
        i. $\langle \overline{BK}_i \rangle \leftarrow MPC.Enc^{GGSW}(\langle \mathbf{s}[i] \rangle, (\langle \overline{\mathbf{s}}_0 \rangle, \ldots, \langle \overline{\mathbf{s}}_{\overline{w}-1} \rangle); XOF, \overline{Q}, \overline{N}, \overline{w}, \overline{\beta_{bk}}, \overline{\nu_{bk}}, flag)$.
        ii. $\overline{BK}_i \leftarrow MPC.Open(\langle \overline{BK}_i \rangle)$.
    (d) Else
        i. $\overline{BK}_i \leftarrow \perp$.
15. If $type = F\text{-}GLWE$ then $\langle \mathbf{s}_{F\text{-}GLWE} \rangle \leftarrow (\langle \mathbf{s}_0[0] \rangle, \ldots, \langle \mathbf{s}_0[N-1] \rangle, \langle \mathbf{s}_1[0] \rangle, \ldots, \langle \mathbf{s}_1[N-1] \rangle, \ldots, \langle \mathbf{s}_{w-1}[0] \rangle, \ldots, \langle \mathbf{s}_{w-1}[N-1] \rangle)$.
16. For $i \in [0, \ldots, \hat{\ell}-1]$ do
    (a) If $type = LWE$ then $\langle PKSK_i \rangle \leftarrow MPC.Enc^{Lev}(\langle \hat{\mathbf{s}}[i] \rangle, \langle \mathbf{s} \rangle; XOF, \beta_{pksk}, Q, \ell, \nu_{pksk}, flag)$.
    (b) Else $\langle PKSK_i \rangle \leftarrow MPC.Enc^{Lev}(\langle \hat{\mathbf{s}}[i] \rangle, \langle \mathbf{s}_{F\text{-}GLWE} \rangle; XOF, \beta_{pksk}, Q, w \cdot N, \nu_{pksk}, flag)$.
    (c) $PKSK_i \leftarrow MPC.Open(\langle PKSK_i \rangle) \pmod{Q}$.
17. If $flag$ then
    (a) $\mathfrak{pk} \leftarrow (\mathfrak{pk}_a, \mathfrak{pk}_b)$.
    (b) $PK \leftarrow (\mathfrak{pk}, \{PKSK_i\}_i, \{KSK_{i,j}\}_{i,j}, \{BK_i\}_i, \{\overline{BK}_i\}_i)$.
18. Else
    (a) $PK_b \leftarrow (\mathfrak{pk}_b, \{PKSK_i\}_i, \{KSK_{i,j}\}_{i,j}, \{BK_i\}_i, \{\overline{BK}_i\}_i)$.
    (b) $PK \leftarrow (seed, PK_b)$.
19. If $\overline{flag}$ then $\langle \mathfrak{sk} \rangle \leftarrow (\langle \mathbf{s} \rangle \pmod{Q}, (\langle \overline{\mathbf{s}}_0 \rangle, \ldots, \langle \overline{\mathbf{s}}_{\overline{w}-1} \rangle))$, else $\langle \mathfrak{sk} \rangle \leftarrow (\langle \mathbf{s} \rangle, \perp)$.
20. If FFT optimizations are to be applied, then the data in BK could be translated into the Fourier domain.
21. Return $(PK, \langle \mathfrak{sk} \rangle)$.

Figure 81: TFHE Threshold Key Generation.

phase) the "online" phase can be done in three rounds (one to open the $seed_i$ values, one for the batched multiplications, and one for the batched opening operations of the data in the public key).

For efficiency one could also pre–process the shared random bits into the shared random values

from the relevant $TUniform$ distributions in a pre–processing phase. Our implementation indeed does this to avoid a huge memory overhead, and to aid deployment. We will discuss this later in more detail in Section 9.6.14.

7.5.4.2 Threshold Decryption Method 1: This method is executed when we set $\overline{flag} = true$ in the key generation methodology. Recall at this point we hold a secret key in secret shared form $\overline{\mathbf{s}} = (\overline{\mathbf{s}}_0, \ldots, \overline{\mathbf{s}}_{\overline{w}-1})$ modulo $\overline{Q}$, and has LWE dimension $\overline{\ell} = \overline{w} \cdot \overline{N}$. The input ciphertext $\mathfrak{ct} = (\mathbf{a}, b)$ is one with respect to modulo Q, and has LWE dimension ℓ, and has noise bound $B_{\mathfrak{ct}}$, i.e.

$$\| b - \mathbf{a} \cdot \mathbf{s} - \Delta \cdot m \quad (\mathrm{mod}\ Q) \| < B_{\mathfrak{ct}}$$

for $\Delta = Q/P$ and an underlying secret key $\mathbf{s} \in \{0, 1\}^{\ell}$.

The $TFHE.SwitchSquash$ operation maps the input ciphertext into a ciphertext $\overline{\mathfrak{ct}} = (\overline{\mathbf{a}}, \overline{b})$ with LWE dimension $\overline{\ell} = \overline{w} \cdot \overline{N}$, modulus $\overline{Q}$ and noise bound $B_{SwitchSquash}$, i.e.

$$\| \overline{b} - \overline{\mathbf{a}} \cdot \overline{\mathbf{s}} - \overline{\Delta} \cdot m \quad (\mathrm{mod}\ \overline{Q}) \| < B_{SwitchSquash},$$

where $\overline{\Delta} = \overline{Q}/P$. The key condition on the output of $TFHE.SwitchSquash$ is that

$$2 \cdot nSmallBnd \cdot 2^{stat} \cdot B_{SwitchSquash} < \frac{\overline{\Delta}}{2}.$$

This means there is enough "noise gap" between the noise term in $\overline{\mathfrak{ct}}$ and the "modulus" $\overline{\Delta}$ in order to apply noise–flooding to the pre–decryption. The noise–flooding can then be applied, using the $PRSS\text{-}Mask$ operations in threshold profile $nSmall$, or using a series of random bits in threshold profile $nLarge$, as described in Figure 82.

TFHE Threshold Decryption – V1

$TFHE.Threshold\text{-}Dec\text{-}1(\mathfrak{ct} = (\mathbf{a}, b), PK, \langle \overline{\mathbf{s}} \rangle)$:
  We use the shorthand $\langle \overline{\mathbf{s}} \rangle = (\langle \overline{\mathbf{s}}_0 \rangle, \ldots, \langle \overline{\mathbf{s}}_{\overline{w}-1} \rangle)$.
   1. If $PK = (seed, PK_b)$ then
       (a) $(\mathfrak{pf}_a, \{KSK^a_{i,j}\}_{i,j}, \{BK^a_i\}_i, \{\overline{BK}^a_i\}_i) \leftarrow TFHE.Expand(seed, \ell, N, w, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w},$
          $\overline{\beta_{bk}}, Q, true)$.
       (b) Parse PK_b as $(*, *, *, \overline{BK}^b)$.
       (c) $\overline{BK} \leftarrow (\{\overline{BK}^a_i\}_i, \overline{BK}^b)$.
   2. Else
       (a) Parse $PK = (*, *, *, *, \overline{BK})$.
       (b) If $\overline{BK} = \{\bot\}_i$ then $abort$.
   3. $\overline{\mathfrak{ct}} = (\overline{\mathbf{a}}, \overline{b}) \leftarrow TFHE.SwitchSquash(\mathfrak{ct}, \overline{BK})$.
   4. $\langle p \rangle \leftarrow \overline{b} - \overline{\mathbf{a}} \cdot \langle \overline{\mathbf{s}} \rangle$.
   5. If threshold profile $nSmall$ then
       (a) $\langle t \rangle \leftarrow PRSS\text{-}Mask.Next(B_{SwitchSquash}, stat)$.
   6. Else
       (a) Set $b_d \leftarrow stat + \lceil \log_2 B_{SwitchSquash} \rceil$.
       (b) $\langle t \rangle \leftarrow MPC.TUniform(1, -2^{b_d}, 2^{b_d}) + MPC.TUniform(1, -2^{b_d}, 2^{b_d})$.
   7. $\langle c \rangle \leftarrow \langle p \rangle + \langle t \rangle$.
   8. $c \leftarrow MPC.Open(\langle c \rangle)$.
   9. Write c as $\overline{\Delta} \cdot m + e \pmod{\overline{Q}}$; this is done by setting $m \leftarrow \lfloor c/\overline{\Delta} \rceil$ and $e \leftarrow c - m \cdot \overline{\Delta}$.
   10. Return m.

Figure 82: TFHE Threshold Decryption – Version 1.

This distributed decryption protocol is secure, since the opening of the value c leaks no information

about the noise value inside $\overline{\mathfrak{ct}}$, since either the addition of the value returned by the *PRSS-Mask.Next* function "floods" the value, or the addition of the two uniform distributions "floods" the value. This follows from Theorem 6 or Lemma 18; but we will provide a full simulation proof later.

For correctness note that the value e has norm bounded by (with overwhelming probability) either by (for threshold profiles *nSmall*)

$$B_{SwitchSquash} \cdot \left(1 + 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat}\right) < 2 \cdot nSmallBnd \cdot 2^{stat} \cdot B_{SwitchSquash} < \frac{\overline{\Delta}}{2}.$$

by (for threshold profiles *nLarge*)

$$B_{SwitchSquash} \cdot \left(1 + 2 \cdot 2^{stat}\right) < 2 \cdot nSmallBnd \cdot 2^{stat} \cdot B_{SwitchSquash} < \frac{\overline{\Delta}}{2}.$$

Thus there is no wrap–around error introduced by adding on the value t, and thus the message is not damaged by the noise flooding.

7.5.4.3 Threshold Decryption Method 2: Our second threshold decryption method for TFHE as– sumes the secret key **s** has been shared with respect to a secret sharing scheme modulo $Q = 2^{64}$. The idea behind this variant is that we use our generic MPC engine in order to extract the bits of the pre–decryption in secret shared form. Thus we can output the message part of the pre–decryption directly, and have no need to worry about noise flooding. The disadvantage of this method is that the methods to extract secret shared bits have relatively large round complexity (for $\log_2 Q = 64$ this is 16 rounds). Hence, this method may be prohibitively expensive on WAN type networks.

Our bit manipulation methods are based on standard techniques to be found in [CdH10], [DFK+06], [NO07] and [sec09]. In the algorithms in Figure 83 and Figure 84 we assume K is a power of two for ease of exposition. In fact the hassle of dealing with non–power of two K is probably not worth it, and vectors can be padded to being of length a power of two. The method used to define *BitDec* for $q = 2^K$ is then immediate, see Figure 84; this method uses the bit-manipulation algorithms defined in Figure 83.

In these techniques a binary adder is required to perform one addition mod $q = 2^K$, with no carry output (i.e. the input is two integers of K bits in length, and the output is also of K bits in length). For ease of implementation and efficiency we chose the Kogge–Stone carry look–ahead adder [KS73] with its iterative version described in Algorithm 1 of [CGTV15]. We describe the adder in Figure 83, which has a total of $2 \cdot \log_2 K + 1$ communication rounds and requires $K + 2 \cdot K \cdot \log_2 K$ triples. In Figure 83 the operation $\langle \mathbf{a} \rangle \ll j$, for $0 \le j < K$, is the operation which takes as input the secret shared vector $(\langle a_i \rangle)_{i=0}^{K-1}$ and outputs the vector $(\langle 0 \rangle)_{i=0}^{j-1}, (\langle a_i \rangle)_{i=0}^{K-j}$.

Given the ability to bit decompose a secret shared value modulo q, it is easy to see how to perform threshold decryption for TFHE, see Figure 85. Recall we have $Q = 2^K$ and $P = 2^{\mathfrak{p}}$; typically $K = 64$ and $\mathfrak{p} = 3$ or 5.

MPC Bit Manipulation Algorithms

$XOR(\langle a \rangle, \langle b \rangle)$:
      1. Return $\langle a \rangle + \langle b \rangle - 2 \cdot \langle a \rangle \cdot \langle b \rangle$.

$BitAdd\left(\langle \mathbf{a} \rangle = (\langle a_i \rangle)_{i=0}^{K-1}, \langle \mathbf{b} \rangle = (\langle b_i \rangle)_{i=0}^{K-1}\right)$: Here bit zero is the least significant bit, and the adaption to when one set of inputs are in the clear is immediate (so we do not describe it). Note in the following we process a vector of secret shared bits at each stage, thus the bit operations (shifting, XOR, Mult etc) correspond to parallel operations on these vectors.
      1. $\langle \mathbf{s} \rangle \leftarrow XOR(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle)$.
      2. $\langle \mathbf{p} \rangle \leftarrow \langle \mathbf{s} \rangle$.
      3. $\langle \mathbf{g} \rangle \leftarrow MPC.Mult(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle)$.
      4. For $i \in [0, \ldots, \log_2 K - 1]$ do
          (a) $\langle \mathbf{t} \rangle \leftarrow MPC.Mult(\langle \mathbf{p} \rangle, \langle \mathbf{g} \rangle \ll 2^i)$.
          (b) $\langle \mathbf{g} \rangle \leftarrow XOR(\langle \mathbf{t} \rangle, \langle \mathbf{g} \rangle)$.
          (c) $\langle \mathbf{p} \rangle \leftarrow MPC.Mult(\langle \mathbf{p} \rangle, (\langle \mathbf{p} \rangle \ll 2^i))$.
      5. $\langle \mathbf{s} \rangle \leftarrow XOR(\langle \mathbf{s} \rangle, \langle \mathbf{g} \rangle \ll 1)$.
      6. Return $(\langle s_i \rangle)_{i=0}^{K-1} = \langle \mathbf{s} \rangle$.

$BitSum\left((\langle a_i \rangle)_{i=0}^{K-1}\right)$:
      1. $\langle s \rangle \leftarrow 0$.
      2. For $i \in [0, \ldots, K-1]$ do
          (a) $\langle s \rangle \leftarrow \langle s \rangle + 2^i \cdot \langle a_i \rangle$.
      3. Return $\langle s \rangle$.

Figure 83: MPC Bit Manipulation Algorithms.

MPC Bit Decomposition

$BitDec(\langle a \rangle)$:
      1. $(\langle r_i \rangle)_{i=0}^{K-1} \leftarrow MPC.GenBits(K)$.
      2. $\langle r \rangle \leftarrow BitSum((\langle r_i \rangle)_{i=0}^{K-1})$.
      3. $\langle t \rangle \leftarrow \langle a \rangle - \langle r \rangle$.
      4. $t \leftarrow MPC.Open(\langle t \rangle)$.
      5. Write $t = \sum_i 2^i \cdot t_i$ for $t_i \in \{0, 1\}$.
      6. $(\langle b_i \rangle)_{i=0}^{K-1} \leftarrow BitAdd\left((t_i)_{i=0}^{K-1}, (\langle r_i \rangle)_{i=0}^{K-1}\right)$.
      7. Return $(\langle b_i \rangle)_{i=0}^{K-1}$.

Figure 84: MPC Bit Decomposition.

TFHE Threshold Decryption – V2

$TFHE.Threshold\text{-}Dec\text{-}2(\mathfrak{ct} = (\mathbf{a}, b), \langle \mathbf{s} \rangle, PK)$:

1. If $ctType(\mathfrak{ct}) = F\text{-}GLWE$ then
   (a) $\mathfrak{ct} = (\mathbf{a}, b) \leftarrow TFHE.KeySwitch(\mathfrak{ct}, KSK; Q, \ell, N, w, \beta_{ksk}, \nu_{ksk})$.
2. $\langle p \rangle \leftarrow b - \mathbf{a} \cdot \langle \mathbf{s} \rangle$.
3. $\langle p \rangle \leftarrow \langle p \rangle + \Delta/2$.
4. $(\langle b_i \rangle)_{i=0}^{K-1} \leftarrow BitDec(\langle p \rangle)$.
5. $\langle m \rangle \leftarrow \sum_{i=0}^{\mathfrak{p}-1} \langle b_{K-1-\mathfrak{p}+i} \rangle \cdot 2^i$.
6. $\langle m \rangle \leftarrow \langle m \rangle - \langle b_{K-1} \rangle \cdot \langle m \rangle$.
   If the error is negative (i.e. $b_{K-1} = 1$) this produces a sharing of zero, otherwise if produces a sharing of m.
7. $m \leftarrow MPC.Open(\langle m \rangle)$.
8. Return m.

Figure 85: TFHE Threshold Decryption – Version 2.

7.5.5 Resharing

Following Design Decision 2 we present methodologies to perform re-sharing of the underlying shared FHE secret key $\langle \mathfrak{sf} \rangle$ in all situations. To ensure pro-active security the method for secret sharing needs to be combined with a mechanism for securely erasing the previous shared key. We denote such an erasure operation by $Erase(x)$, which means that the party removes x from memory and storage.

We assume the more general case where we wish to transfer the sharing $\langle \mathfrak{sf} \rangle$ from a set of players S_1, of size $\mathfrak{n}_1$, to a set of players S_2 of size $\mathfrak{n}_2$. The maximum number of corrupted parties in each set is denoted by $\mathfrak{t}_1$ and $\mathfrak{t}_2$, and we assume that $\mathfrak{t}_1 < \mathfrak{n}_1/3$ and $\mathfrak{t}_2 < \mathfrak{n}_2/3$.

The resharing protocol is built on top of each set of players running an independent MPC protocol; thus the same protocol works in threshold profiles *nSmall* and *nLarge*. We assume that the secret $\langle \mathfrak{sf} \rangle^{S_1}$ is already shared between the players in S_1, the goal is to obtain a sharing of the same value between the players in S_2, i.e. we aim to obtain $\langle \mathfrak{sf} \rangle^{S_2}$. The case of resharing amongst the same group of players, i.e. $S = S_1 = S_2$, follows from the general case. The protocol is described in Figure 86; here we let q denote whatever modulus is used to share the underlying secret key in the above FHE schemes; i.e. either $q = Q_1$ in the case of BGV/BFV, or $q = Q$ or $q = \overline{Q}$ in the case of TFHE, and we let N denote the dimension of the secret key.

ReShare($S_1, S_2, \langle \mathfrak{sf} \rangle^{S_1}$)

ReShare($S_1, S_2, \langle \mathfrak{sf} \rangle^{S_1}$):
1. The players in S_2 execute $MPC^O.Init()$.
   If $S_1 = S_2$ then this initializes an independent MPC engine from any existing engine amongst the players in S_1.
2. Write $\langle \mathfrak{sf} \rangle^{S_1} = ((s_i)^{S_1})_{i=0}^{N}$ for $s_i \in GR(q, F)$.
3. For $i \in [1, \dots, N]$ and $j \in [1, \dots, \mathfrak{n}_1]$ in parallel execute
   (a) The players in S_2 execute $\langle r_{i,j} \rangle^{S_2} \leftarrow MPC^O.NextRandom()$.
   (b) Execute $r_{i,j} \leftarrow RobustOpen(\mathcal{P}_j, \langle r_{i,j} \rangle^{S_2})$, so that player $\mathcal{P}_j \in S_1$ learns $r_{i,j}$.
   (c) Player $\mathcal{P}_j \in S_1$ computes $v_{i,j} = r_{i,j} + (s_i)_j^{S_1}$.
   (d) If $S_1 = S_2$ then
       i. Player $\mathcal{P}_j$ executes $Synch\text{-}Broadcast(\mathcal{P}_j, v_{i,j})$.
   (e) Else
       i. Player $\mathcal{P}_j \in S_1$ sends $v_{i,j}$ to all players in S_2.
       ii. Players $\mathcal{P}_k \in S_2$ execute $Synch\text{-}Broadcast(\mathcal{P}_k, v_{i,j})$ amongst the players in S_2.
       iii. The players in S_2 take as the agreed value $v_{i,j}$ the value which has the majority vote from the previous step (with some deterministic way of solving a tie).
   (f) Player $\mathcal{P}_j \in S_1$ executes $Erase((s_i)_j^{S_1})$ and $Erase(r_{i,j})$.
   (g) The players in S_2 compute $\langle (s_i)_j^{S_1} \rangle^{S_2} \leftarrow v_{i,j} - \langle r_{i,j} \rangle^{S_2}$.
4. For $i \in [1, \dots, N]$ the players in S_2 compute
   (a) Locally compute a sharing of the syndrome polynomial $\langle S_{\mathbf{e}_i}(Z) \rangle^{S_2}$ for the sharing $(s_i)^{S_1}$ hidden in the shares $\langle (s_i)_j^{S_1} \rangle^{S_2}$.
   (b) $S_{\mathbf{e}_i}(Z) \leftarrow MPC.Open(\langle S_{\mathbf{e}_i}(Z) \rangle^{S_2})$.
   (c) $\mathbf{e}_i \leftarrow SynDecode^{GR}(q, S_{\mathbf{e}_i}(Z))$.
   (d) For $j \in [1, \dots, \mathfrak{n}_1]$ do $\langle (s_i)_j^{S_1} \rangle^{S_2} \leftarrow \langle (s_i)_j^{S_1} \rangle^{S_2} - \mathbf{e}_i^{(j)}$.
   (e) $\langle s_i \rangle^{S_2} \leftarrow \sum_j \langle (s_i)_j^{S_1} \rangle^{S_2} \cdot \delta_j(0)$, where $\delta_j(Z)$ is the Lagrange polynomial for the secret sharing used by the players in S_1.

Figure 86: *ReShare($S_1, S_2, \langle \mathfrak{sf} \rangle^{S_1}$).*

Note in line 3(e)i an adversarial $\mathcal{P}_j$ may not send the same value to each player in S_2. Thus in the next broadcast step we ensure that all honest players agree on the same value $v_{i,j}$ via taking the majority value as the value $v_{i,j}$ in line 3(e)iii. If $\mathcal{P}_j$ is honest then clearly the honest players in S_2 will agree on $v_{i,j}$.

The value computed in line 3g is thus, for honest $\mathcal{P}_j$, a valid secret sharing of $\mathcal{P}_j$'s share value. If $\mathcal{P}_j$ is dishonest then we do not care what value is picked for $v_{i,j}$ as we can correct for it later, as long as the value is consistent amongst all the honest players in S_2. Note the robust open of $r_{i,j}$ allows $\mathcal{P}_j$ to totally masks $\langle s_i \rangle_j$; with *RobustOpen* working as we have $\mathfrak{t}_2 < \mathfrak{n}_2/3$.

The next steps determine which values are actually dishonestly sent. This is done by computing the syndrome polynomial, opening it, and then determining the error locations and error values. That *SynDecode*^{GR} works since $\mathfrak{t}_1 < \mathfrak{n}_1/3$, so only $\mathfrak{t}_1$ errors can exist in total after line 3g. Note, this is the only place where we actually use syndrome decoding in this document.

7.6 Layer Five

Recall from Section 4.11 we present three types of zero–knowledge proofs of correct encryption for our FHE schemes. Two types which is based on elliptic curves, and a third type which is based on MPC–in–the–Head. All types of proof are provide post–quantum zero–knowledge, but the two based on elliptic curves are not post–quantum secure with respect to the soundness property, whereas the latter are. The proofs based on elliptic curves are more suited to TFHE–style FHE, as the elliptic curve needs to be pairing friendly and have group order significantly larger than the FHE ciphertext modulus. This means their application for BGV and BFV style encryption is less appropriate.

7.6.1 Interpreting Encryption as a Subset Sum

Recall from Section 4.11 we phrase two of our zero–knowledge proofs for correct encryption, in terms of showing, in zero–knowledge, the knowledge of a solution to a subset–sum problem. Thus our zero–knowledge proofs need to show knowledge of a witness to a subset–sum problem instance. We deal with a more general subset–sum problem, than that considered normally, in that our subset–sum problem will consist of a set of linear equations, rather than just one.

Definition 10 (Subset–Sum Problem). A (generalized) subset–sum problem is to solve the given linear system

$$\mathbf{s} = \mathbf{A} \cdot \mathbf{b} \pmod{q},$$

where $\mathbf{s} \in (\mathbb{Z}/(q))^d$ and $\mathbf{A} \in (\mathbb{Z}/(q))^{d \times B}$ are known, and $\mathbf{b} \in \{0, 1\}^B$ is unknown.

Such subset–sum problems have two problem parameters,
- d: The subset–sum dimension, i.e. the number of subset–sums in a given problem instance.
- B: The total number of secret bits underlying the statement being proved. The actual values we denote by the vector $\mathbf{b} \in \{0, 1\}^B$.

We now recap on the various encryption methods here and how they relate to these subset–sum parameters.

7.6.1.1 BGV: For BGV we have the linear equations

$$\mathbf{c}_0 = B_{\mathfrak{pk}} \cdot \mathbf{v} + P \cdot \mathbf{e}_0 + \mathbf{m},$$
$$\mathbf{c}_1 = A_{\mathfrak{pk}} \cdot \mathbf{v} + P \cdot \mathbf{e}_1,$$

where $A_{\mathfrak{pk}}$ and $B_{\mathfrak{pk}}$ are $N \times N$ public matrices, $\mathbf{v}$ is a secret binary vector of length N, and $\mathbf{e}_0$ (resp. $\mathbf{e}_1$) is a secret vector of length N, each component of which is derived from a linear combination of $2 \cdot B_{BGV}$ secret bits. Thus we have

$$d = 2 \cdot N$$

linear equations in our subset–sum system in

$$B = N + 2 \cdot (N \cdot 2 \cdot B_{BGV}) + \log_2 P = N \cdot (4 \cdot B_{BGV} + 1) + \log_2 P$$

binary variables.

7.6.1.2 BFV: For BFV we have the linear equations

$$\mathbf{c}_0 = B_{\mathfrak{pk}} \cdot \mathbf{v} + \mathbf{e}_0 + \Delta \cdot \mathbf{m},$$
$$\mathbf{c}_1 = A_{\mathfrak{pk}} \cdot \mathbf{v} + \mathbf{e}_1,$$

where $A_{\mathfrak{pf}}$ and $B_{\mathfrak{pf}}$ are $N \times N$ public matrices, $\mathbf{v}$ is a secret binary vector of length N, and $\mathbf{e}_0$ (resp. $\mathbf{e}_1$) is a secret vector of length N, each component of which is derived from a linear combination of $2 \cdot B_{BGV}$ secret bits. Thus, just as for BGV, we have

$$d = 2 \cdot N$$

linear equations in our subset–sum system in

$$B = N + 2 \cdot (N \cdot 2 \cdot B_{BGV}) + \log_2 P = N \cdot (4 \cdot B_{BGV} + 1) + \log_2 P$$

binary variables.

7.6.1.3 TFHE: For TFHE we have the linear equations

$$\mathbf{a} = A_{\mathfrak{pf}} \cdot \overleftrightarrow{\mathbf{r}} + \mathbf{e}_1,$$
$$b = \mathfrak{pf}_b \cdot \mathbf{r} + e_2 + \Delta \cdot m,$$

where

- $A_{\mathfrak{pf}}$ is an $\hat{\ell} \times \hat{\ell}$ public matrix,
- $\mathbf{r}$ is a secret binary vector of length $\hat{\ell}$,
- $\mathbf{e}_1$ is secret and chosen from $TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$,
- and e_2 is secret and chosen from $TUniform(1, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})$.

Recall $\overleftrightarrow{\mathbf{r}}$ is the vector $\mathbf{r}$ written backwards. Thus we have

$$d = \hat{\ell} + 1$$

linear equations in our subset–sum system in

$$B = \hat{\ell} + (\hat{\ell} \cdot (b_{\hat{\ell}} + 2)) + (b_{\hat{\ell}} + 2) + \log_2 P = \hat{\ell} \cdot (b_{\hat{\ell}} + 3) + (b_{\hat{\ell}} + 2) + \log_2 P$$

binary variables.

7.6.2 Pairing Based Elliptic Curves

The soundness of our vector commitment based proofs are based on the hardness of the discrete logarithm problem given in Definition 1, namely the (m, n)–Discrete Logarithm assumption. Recall, this problem requires a set of pairing based groups $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$. The pairing will be denoted as a map

$$e : \mathbb{G} \times \hat{\mathbb{G}} \longrightarrow \mathbb{G}_T.$$

In our instantiation we will assume that the associated prime group order satisfies

$$r > \max(2^{l(sec)}, 2^{\lceil \log(\bar{B}+1) \rceil + 1} \cdot \bar{q}),$$

for some polynomial $l : \mathbb{N} \to \mathbb{N}$. For security recommendations as to the exact choice of underlying pairing group see Section 8.6.2.

7.6.3 ZKPoKs Based on Vector Commitments

In this section we present our ZKPoKs which are not post–quantum secure with respect to soundness. Recall there are two such approaches, one which treats the input problem as a subset–sum, and produces zero–knowledge proofs with no soundness slack, and another which introduces some soundness slack.

7.6.3.1 Required Hash Functions: Our CRS construction and NIZK proofs use hash functions

$$\mathcal{H}_{vc}, \mathcal{H}_t, \mathcal{H}_\omega : \{0,1\}^* \rightarrow (\mathbb{Z}/(r))^{\bar{B}_1} \qquad \mathcal{H}_{agg}, Hash' : \{0,1\}^* \rightarrow (\mathbb{Z}/(r))^2$$

$$\mathcal{H}_{agg'} : \{0,1\}^* \rightarrow \mathbb{Z}/(r)^7 \qquad \mathcal{H}_{linmap} : \{0,1\}^* \rightarrow (\mathbb{Z}/(r))^{\bar{d}+1}$$

$$\mathcal{H}_\phi, \mathcal{H}_\Xi : \{0,1\}^* \rightarrow (\mathbb{Z}/(r))^{128} \qquad \mathcal{H}_\chi, \mathcal{H}_z : \{0,1\}^* \rightarrow \mathbb{Z}/(r)$$

$$Hash : \{0,1\}^* \rightarrow (\mathbb{Z}/(r))^* \qquad \mathcal{H}_R : \{0,1\}^* \rightarrow \mathcal{D}^{128 \times (\bar{B}_r + \bar{d} + 4)}$$

where $\mathcal{D}$ is a distribution over $\{-1, 0, 1\}$ that outputs zero with probability 1/2 and 1 and -1 with probability 1/4 each. These hash functions are modeled as random oracles, and are explicitly defined in Figure 87. Theoretically, speaking these hash functions are 'chosen' during the CRS generation phase, but in practice they are implemented as in Figure 87, with only a session identifier being chosen during the CRS generation phase. We treat the session identifier as an implicit input to the hash functions in order not to overload notation. Practically, the same hash function is used, i.e., *SHAKE*-256 in XOF mode. The session identifier is also acting as domain separator, since some of these functions have the same inputs. Then, for each of the hash functions we use *DSep*(*X*), where *X* is randomly chosen during the CRS generation. In what follows, for a hash function $\mathcal{H}_y$ we simply denote by *DSep*(*ZKHY*) the associated domain separator, where *ZKHY* stands for "ZKPoKs Hash function Y". If the hash function name is too long, the name string is truncated to fit into 5 characters. The inputs of the hash functions are abstracted, soberly written as *x* but fully detailed in the scheme descriptions. In the algorithms related to the ZKPoKs, the concatenation of inputs is denoted by a comma, e.g., *x, y* is equivalent to *x*||*y*. Regarding the implementation, all input data are concatenated as list of bytes, and, as usual, the value of *dist* = 128 is used within the calls to *XOF.Next*. Note that the implementation of *Hash* is the same as $\mathcal{H}_\chi$ and $\mathcal{H}_z$ because the probability of obtaining 0 as an output is negligible.

Hash Functions for VC–based Proofs

$\mathcal{H}_{vc}(x),\ \boxed{\mathcal{H}_t(x)}\ ,\ \overset{.\,.\,.\,.}{\underset{.\,.\,.\,.}{\mathcal{H}_\omega(x)}}$:

    1. $XOF.Init(x, DSep(ZKHVC)\ \boxed{DSep(ZKHT)}\ \overset{.\,.}{\underset{.\,.}{DSep(ZKHW)}}\ )$; i.e. use $DSep(ZKHVC)$ for
        function $\mathcal{H}_{vc}$, $DSep(ZKHT)$ for function $\mathcal{H}_t$ and $DSep(ZKHW)$ for function $\mathcal{H}_\omega$.
    2. Return $XOF.Next(\bar{B}_1, r)$

$\mathcal{H}_{agg}(x),\ \boxed{Hash'}$:

    1. $XOF.Init(x, DSep(ZKHAG)\ \boxed{DSep(ZKH')}\ )$; i.e. use $DSep(ZKHAG)$ for function $\mathcal{H}_{agg}$
        and $DSep(ZKH')$ for function $Hash'$.
    2. Return $XOF.Next(2, r)$

$\mathcal{H}_{agg'}(x)$:
    1. $XOF.Init(x, DSep(ZKHA'))$
    2. Return $XOF.Next(7, r)$

$\mathcal{H}_{linmap}(x)$:
    1. $XOF.Init(x, DSep(ZKHLI))$
    2. Return $XOF.Next(\bar{d} + 1, r)$

$\mathcal{H}_\phi(x),\ \boxed{\mathcal{H}_\xi(x)}$:

    1. $XOF.Init(x, DSep(ZKHPH)\ \boxed{DSep(ZKHXI)}\ )$; i.e. use $DSep(ZKHPH)$ for function $\mathcal{H}_\phi$ and
        $DSep(ZKHXI)$ for function $\mathcal{H}_\xi$.
    2. Return $XOF.Next(128, r)$

$\mathcal{H}_\chi(x),\ \boxed{\mathcal{H}_z(x)}\ ,\ \overset{.\,.\,.\,.}{\underset{.\,.\,.\,.}{Hash(x)}}$:

    1. $XOF.Init(x, DSep(ZKHCH)\ \boxed{DSep(ZKHZ)}\ \overset{.\,.}{\underset{.\,.}{DSep(ZKHW)}}\ )$; i.e. use $DSep(ZKHCH)$ for
        function $\mathcal{H}_\chi$ $DSep(ZKHZ)$ for function $\mathcal{H}_z$, and $DSep(ZKH)$ for $Hash(x)$.
    2. Return $XOF.Next(1, r)$

$\mathcal{H}_R(x)$: Hash x into a vector in $\mathcal{D}^{128 \times (\bar{B}_r + \bar{d} + 4)}$
    1. $XOF.Init(x, DSep(ZKHR))$
    2. $o \leftarrow [\,]$
    3. For $i \in [1, 128]$
        (a) $h \leftarrow XOF.Next(2 \times (\bar{B}_r + \bar{d}) + 4)$
        (b) Treating h as a bitstring, For odd $j \in [1, (\bar{B}_r + \bar{d}) + 4]$,
            i. If h[j]||h[j+1] = '00' or '01', $v \leftarrow 0$
            ii. Else if h[j]||h[j+1] = '10', $v \leftarrow 1$
            iii. Else $v \leftarrow (-1)$
        (c) $o[i] \leftarrow o[i]||v$
    4. Return o

Figure 87: Hash functions for use in the VC–based ZKPoKs.

7.6.3.2 CRS and the CRS SetUp: Both of our vector commitment based methods make use of the same structured CRS generation method, given in Figure 88 and Figure 89. Several distributed protocols can be used to generate the structured CRS without relying on a centralized trusted party. Among them, the ceremony protocols of [NRBB24, KMSV21] make it possible for K parties to sequentially contribute to the secret trapdoor α, which is a obtained as a product $\alpha = \prod_{i=1}^{K} \tau_i$ of individual randomizers $\{\tau_i\}_{i=1}^{K}$.

> **CRS Generation**
>
> *CRS-Gen*(*sec*, *params*):
> On input of a security parameter *sec* and public parameters *params* consisting of either
> $\{\bar{q}, \bar{d}, \bar{B}\}$ (in the case of Type–1 proofs) or $\{\bar{q}, \bar{d}, \bar{B}_l, \bar{B}_r, \bar{U}_\infty\}$ (in the case of Type–2 proofs)
> where $\bar{d}, \bar{B}, \bar{q}$ are upper bounds for d, B and q, whereas $\bar{U}_\infty, \bar{B}_l$ and $\bar{B}_r$ are upper bounds for
> U_∞, B_l and B_r conduct the following steps to generate the CRS **p**.
>   1. Generate pairing–friendly groups $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$ of prime order $r >$
> $\max(2^{l(sec)}, 2^{\lceil \log(\bar{B}+1) \rceil+1} \bar{q})$, for some polynomial $l : \mathbb{N} \to \mathbb{N}$.
>   2. If using the first type of vector commitment based proofs then $\bar{B}_1 \leftarrow \bar{B} + \bar{d} \cdot (1 + \lfloor \log(\bar{B} +$
> $1) \rfloor)$. If using the second type of VC–based proofs, set $\bar{U}_2 \triangleq \bar{U}_\infty \cdot \sqrt{\bar{B}_r}$, $\bar{U}_k \triangleq (\bar{B}_l + \bar{B}_r)/2$,
> $\bar{U}_H \triangleq 13.32 \cdot \sqrt{\bar{U}_2^2 + \bar{U}_k^2} \cdot \bar{d}$ and $\bar{m} = \lceil 1 + \log \bar{U}_H \rceil$. Then, set $\bar{B}_1 \leftarrow \max(\bar{B}_l + 128 \cdot \bar{m}, \bar{B}_r +$
> $\bar{d} + 4)$.
>   3. Run the distributed ceremony protocol of Figure 89 to jointly generate a secret random
> $\alpha \leftarrow \mathbb{Z}/(r)$ and public $g_1, \ldots, g_{\bar{B}_1}, g_{\bar{B}_1+2}, \ldots, g_{2\bar{B}_1} \in \mathbb{G}$ as well as $\hat{g}_1, \ldots, \hat{g}_{\bar{B}_1} \in \hat{\mathbb{G}}$, where
> $g_i = g^{(\alpha^i)}$ for each $i \in [2\bar{B}_1] \setminus \{\bar{B}_1 + 1\}$ and $\hat{g}_i = \hat{g}^{(\alpha^i)}$ for each $i \in [\bar{B}_1]$.
>   4. Return the common reference string
>
> $$\mathbf{p} = \left((\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T), g, \hat{g}, \{g_i\}_{i \in [2\bar{B}_1] \setminus \{\bar{B}_1+1\}}, \{\hat{g}_i\}_{i \in [\bar{B}_1]} \right).$$

Figure 88: CRS Generation for the Vector–Commitment–Based NIZK.

CRS-Gen.Init$((\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T), g, \hat{g}, \mathbf{H})$:

    The initial state after round zero is the default CRS pp_0

$$pp_0 \leftarrow \big(g_{1,0} = g, \quad g_{2,0} = g, \quad \ldots, \quad g_{\bar{B}_1,0} = g, \quad g_{\bar{B}_1+2,0} = g, \ldots, \quad g_{2\bar{B}_1,0} = g$$
$$\hat{g}_{1,0} = \hat{g}, \quad \hat{g}_{2,0} = \hat{g}, \quad \ldots, \quad \hat{g}_{\bar{B}_1,0} = \hat{g}\,\big)$$

CRS-Gen.Update(pp_{j-1}):

    At the beginning of round j, the current CRS pp_{j-1} is assumed to be

$$pp_{j-1} = \big(g_{1,j-1} = g^{\alpha_{j-1}}, \quad g_{2,j-1} = g^{(\alpha_{j-1}^2)}, \quad \ldots, \quad g_{\bar{B}_1,j-1} = g^{(\alpha_{j-1}^{\bar{B}_1})},$$
$$g_{\bar{B}_1+2,j-1} = g^{(\alpha_{j-1}^{\bar{B}_1+2})}, \quad \ldots, \quad g_{2\bar{B}_1,j-1} = g^{(\alpha_{j-1}^{2\bar{B}_1})},$$
$$\hat{g}_{1,j-1} = \hat{g}^{\alpha_{j-1}}, \quad \hat{g}_{2,j-1} = \hat{g}^{(\alpha_{j-1}^2)}, \quad \ldots, \quad \hat{g}_{\bar{B}_1,j-1} = \hat{g}^{(\alpha_{j-1}^{\bar{B}_1})}\,\big)$$

1. The j–th contributor chooses a random $\tau_j \leftarrow \mathbb{Z}/(r)$ and computes

$$pp_j \leftarrow \big(\ g_{1,j} = g^{\tau_j \alpha_{j-1}}, \quad g_{2,j} = g^{(\tau_j^2 \alpha_{j-1}^2)}, \quad \ldots, \quad g_{\bar{B}_1,j} = g^{(\tau_j^{\bar{B}_1} \alpha_{j-1}^{\bar{B}_1})},$$
$$g_{\bar{B}_1+2,j} = g^{(\tau_j^{\bar{B}_1+2} \alpha_{j-1}^{\bar{B}_1+2})}, \quad \ldots, \quad g_{2\bar{B}_1,j} = g^{(\tau_j^{2\bar{B}_1} \alpha_{j-1}^{2\bar{B}_1})},$$
$$\hat{g}_{1,j} = \hat{g}^{\tau_j \alpha_{j-1}}, \quad \hat{g}_{2,j} = \hat{g}^{(\tau_j^2 \alpha_{j-1}^2)}, \quad \ldots, \quad \hat{g}_{\bar{B}_1,j} = \hat{g}^{(\tau_j^{\bar{B}_1} \alpha_{j-1}^{\bar{B}_1})}\ \big),$$

    which implicitly defines $\alpha_j = \tau_j \cdot \alpha_{j-1}$.

2. Then, the contributor proves knowledge of $\tau_j \in \mathbb{Z}/(r)$ such that $g_{1,j} = g_{1,j-1}^{\tau_j}$. This non–interactive proof $\bar{\pi}_{PoK,j} = (h_{PoK,j}, s_{PoK,j}) \in (\mathbb{Z}/(r))^* \times \mathbb{Z}/(r)$ is obtained by choosing $r_{PoK,j} \leftarrow \mathbb{Z}/(r)$, computing $R_{PoK,j} \leftarrow g_{1,j-1}^{r_{PoK,j}}, h_{PoK,j} \leftarrow$ *Hash*$(g_{1,j}, g_{1,j-1}, R_{PoK,j})$, and

$$s_{PoK,j} \leftarrow r_{PoK,j} + h_{PoK,j} \cdot \tau_j \bmod r.$$

3. All parties verify the proof $\bar{\pi}_{PoK_j} = (h_{PoK,j}, s_{PoK,j})$ by testing the equality

$$h_{PoK,j} = Hash\big(g_{1,j}, \ g_{1,j-1}, \ g_{1,j-1}^{s_{PoK,j}} \cdot g_{1,j}^{-h_{PoK,j}}\big) \tag{34}$$

    and reject pp_j if (34) does not hold or if $g_{1,j} = 1_{\mathbb{G}}$.

4. All parties then compute $(\rho_1, \rho_2) \leftarrow$ *Hash*$'(pp_j) \in (\mathbb{Z}/(r))^2$ and reject pp_j if one of the equalities

$$e\Big(\prod_{\substack{i=1 \\ i \neq \bar{B}_1+1 \\ i \neq \bar{B}_1+2}}^{2\bar{B}_1} g_{i,j}^{(\rho_1^{i-1})}, \ \hat{g} \cdot \prod_{\ell=1}^{\bar{B}_1-1} \hat{g}_{\ell,j}^{(\rho_2^{\ell})} \Big) = e\Big(g \cdot \prod_{\substack{i=1, \\ i \neq \bar{B}_1, \\ i \neq \bar{B}_1+1}}^{2\bar{B}_1-1} g_{i,j}^{(\rho_1^{i})}, \ \prod_{\ell=1}^{\bar{B}_1} \hat{g}_{\ell,j}^{(\rho_2^{\ell-1})} \Big) \tag{35}$$
$$e(g_{\bar{B}_1+2,j}, \hat{g}) = e(g_{\bar{B}_1,j}, \hat{g}_{2,j})$$

    does not hold.

5. If the previous checks are all satisfied, then all parties accept pp_j as the updated CRS after round j.

CRS-Gen.Output():

    At the end of round K, the parties output $pp \leftarrow pp_j$, where $j \in [K]$ is the largest index such that pp_j was accepted by all parties.

Figure 89: Ceremony for the CRS Generation.

7.6.3.3 Method 1 Proof Construction: Our first, subset–sum based, approach is similar to that of Del Pino *et al.* [dPLS19], and it appeared in [Lib24]. Namely, we re–write the subset sum relation over the integers as an equation

$$\mathbf{s} = \mathbf{A} \cdot \mathbf{b} - \mathbf{k} \cdot q \tag{36}$$

for some vector $\mathbf{k} \in \mathbb{Z}^d$ of infinity norm $\|\mathbf{k}\|_\infty \leq (B+1)/2 < 2^{\lfloor \log(B+1) \rfloor}$. Note that both members of (36) are bounded by $(B+1) \cdot q$ in infinity norm. So, if we prove that (36) holds modulo a sufficiently large prime r (i.e., such that $r > 2^{\lceil \log(B+1) \rceil + 1} q$), we also prove that it holds over the integers. To prove (36) over $\mathbb{Z}$, we will prove that $\mathbf{b} \in \{0, 1\}^B$ and $\mathbf{k} \in [-2^{\lfloor \log(B+1) \rfloor}, 2^{\lfloor \log(B+1) \rfloor} - 1]^d$. The respective prover and verifier are given in Figure 90 and Figure 91 respectively.

VC-Prove-1

VC-Prove-1$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$:

Where $\mathbf{p}$ is the CRS, $\mathbf{s} \in (\mathbb{Z}/(q))^d$ and $\mathbf{A} \in (\mathbb{Z}/(q))^{d \times B}$ are the public subset–sum statement which, along with $q \leq \bar{q}, d \leq \bar{d}$, and $B \leq \bar{B}$ form the complete statement $\mathbf{x}$, and $\mathbf{b} \in \{0, 1\}^B$ is the private witness. The bounds $\bar{q}, \bar{d}$ and $\bar{B}$ are in the CRS $\mathbf{p}$. For the proof we think of $\mathbf{s}$ and $\mathbf{A}$ consisting of elements in $\mathbb{Z}/(r)$.

1. Compute $\mathbf{k} \in [-\frac{B+1}{2}, \frac{B+1}{2}]^d$ satisfying (36) and let $\tilde{\mathbf{k}} \leftarrow \mathbf{g}_{1+\lfloor \log(B+1) \rfloor}^{-1}(\mathbf{k}) \in \{0, 1\}^{d \cdot (1+\lfloor \log(B+1) \rfloor)}$.

2. Commit to $\tilde{\mathbf{w}} = (\mathbf{b}^\top \mid \tilde{\mathbf{k}}^\top \mid \mathbf{0}^{\bar{B}_1 - B - d \cdot (1+\lfloor \log(B+1) \rfloor)})^\top \in \{0, 1\}^{\bar{B}_1}$ by choosing $\gamma \leftarrow \mathbb{Z}/(r)$ and computing $\hat{C} \leftarrow \hat{g}^\gamma \cdot \prod_{j=1}^{\bar{B}_1} \hat{g}_j^{w_j}$.

3. Compute $\mathbf{y} = (y_1, \ldots, y_{\bar{B}_1}) \leftarrow \mathcal{H}_{vc}(\mathbf{x}, \hat{C}) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.

4. Choose $\gamma_y \leftarrow \mathbb{Z}/(r)$ and compute

$$C_y \leftarrow g^{\gamma_y} \cdot \prod_{j=1}^{\bar{B}_1} g_{\bar{B}_1 + 1 - j}^{y_j \cdot w_j}$$

5. Compute $\mathbf{t} = (t_1, \ldots, t_{\bar{B}_1}) \leftarrow \mathcal{H}_t(\mathbf{y}, \mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.

6. Compute $\bar{\mathbf{h}} \leftarrow \mathcal{H}_{linmap}(\mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^{\bar{d}+1}$ and let $\mathbf{h} \in (\mathbb{Z}/(r))^{d+1}$ be the first $d + 1$ entries of $\bar{\mathbf{h}}$.

7. Compute $(\delta_{eq}, \delta_y) \leftarrow \mathcal{H}_{agg}(\mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^2$.

8. Parse $\mathbf{h}$ as $\mathbf{h} = (\mathbf{h}_0^\top \mid \delta_{\mathbf{h}})^\top$, with $\mathbf{h}_0 \in (\mathbb{Z}/(r))^d$.

9. Compute $t_{\mathbf{h}} \leftarrow \mathbf{h}_0^\top \cdot \mathbf{s} \bmod p$ and $\boldsymbol{\alpha}_{\mathbf{h}}^\top \leftarrow \mathbf{h}_0^\top \cdot \mathbf{A}_1 \bmod r$, where $\mathbf{A}_1 \in \mathbb{Z}^{d \times \bar{B}_1}$ is defined in (38).

10. Compute the polynomial $P_\pi[X] = \sum_{i=0}^{2\bar{B}_1} \nu_i \cdot X^i$ given by

$$P_\pi[X] \leftarrow \left(\delta_y \cdot \gamma_y + \sum_{i=1}^{\bar{B}_1} (\delta_y \cdot y_i \cdot w_i + (\delta_{eq} \cdot t_i - \delta_y) \cdot y_i + \delta_{\mathbf{h}} \cdot \boldsymbol{\alpha}_{\mathbf{h}}[i]) \cdot X^{\bar{B}_1 + 1 - i} \right)$$

$$\cdot (\gamma + \sum_{i=1}^{\bar{B}_1} w_i \cdot X^i)$$

$$- \left(\gamma_y + \sum_{i=1}^{\bar{B}_1} y_i \cdot w_i \cdot X^{\bar{B}_1 + 1 - i} \right) \cdot \left(\sum_{i=1}^{\bar{B}_1} \delta_{eq} \cdot t_i \cdot X^i \right) - t_{\mathbf{h}} \cdot \delta_{\mathbf{h}} \cdot X^{\bar{B}_1 + 1}.$$

11. Compute $\pi \leftarrow g^{\nu_0} \cdot \prod_{i=1, i \neq \bar{B}_1 + 1}^{2\bar{B}_1} g_i^{\nu_i}$.

12. Return the proof $prf = (\hat{C}, C_y, \pi) \in \hat{\mathbb{G}} \times \mathbb{G}^2$.

Figure 90: The First Vector–Commitment–Based NIZK Prover.

To understand, and justify the correctness, of this proof system we first need to set up some no–

> **VC-Verify-1**
>
> $\textit{VC-Verify-1}(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \textit{prf})$:
>   Given a statement $(\mathbf{A}, \mathbf{s})$, along with $q \le \bar{q}, d \le \bar{d}$, and $B \le \bar{B}$ to form the complete statement
> $\mathbf{x}$ and a purported proof $\textit{prf} = (\hat{C}, C_y, \pi) \in \hat{\mathbb{G}} \times \mathbb{G}^2$.
>   1. $\mathbf{y} = (y_1, \ldots, y_{\bar{B}_1}) \leftarrow \mathcal{H}_{vc}(\mathbf{x}, \hat{C}) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.
>   2. $\mathbf{t} = (t_1, \ldots, t_{\bar{B}_1}) \leftarrow \mathcal{H}_t(\mathbf{y}, \mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.
>   3. $\bar{\mathbf{h}} \leftarrow \mathcal{H}_{linmap}(\mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^{d+1}$.
>   4. $(\delta_{eq}, \delta_y) \leftarrow \mathcal{H}_{agg}(\mathbf{x}, \hat{C}, C_y) \in (\mathbb{Z}/(r))^2$.
>   5. Let $\mathbf{h} \in (\mathbb{Z}/(r))^{d+1}$ be the first $d+1$ elements of $\bar{\mathbf{h}}$.
>   6. Parse $\mathbf{h}$ as $\mathbf{h} = (\mathbf{h}_0^\top \mid \delta_{\mathbf{h}})^\top$, with $\mathbf{h}_0 \in (\mathbb{Z}/(r))^d$.
>   7. Compute $t_{\mathbf{h}} \leftarrow \mathbf{h}_0^\top \cdot \mathbf{s} \bmod p$ and $\boldsymbol{\alpha}_{\mathbf{h}}^\top \leftarrow \mathbf{h}_0^\top \cdot \mathbf{A}_1 \bmod r$.
>   8. Return one if the following equality holds and zero otherwise:
>
> $$e(\pi, \hat{g}) = e\Big(C_y^{\delta_y} \cdot \prod_{i=1}^{\bar{B}_1} g_{\bar{B}_1+1-i}^{(\delta_{eq} \cdot t_i - \delta_y) \cdot y_i + \delta_{\mathbf{h}} \cdot \boldsymbol{\alpha}_{\mathbf{h}}[i]}, \hat{C}\Big) \cdot e\Big(C_y, \prod_{i=1}^{\bar{B}_1} \hat{g}_i^{\delta_{eq} \cdot t_i}\Big)^{-1} \cdot e(g_1, \hat{g}_{\bar{B}_1})^{-t_{\mathbf{h}} \cdot \delta_{\mathbf{h}}}.$$

Figure 91: The First Vector–Commitment–Based NIZK Verifier.

tation: For any integer $z \in \mathbb{Z}$, we define $\mathbf{g} = (1, 2, 4, \ldots, 2^{z-2}, -2^{z-1})^\top \in \mathbb{Z}^{1 \times z}$ so that any integer $v \in [-2^{z-1}, 2^{z-1} - 1]$ can be decomposed as binary vector $\mathbf{v} = \mathbf{g}^{-1}(v) \in \{0, 1\}^z$ such that $v = \mathbf{g}^\top \cdot \mathbf{v}$. The prover will then prove knowledge of a binary vector $(\mathbf{b}^\top \mid \tilde{\mathbf{k}}^\top)^\top \in \{0, 1\}^{B+d \cdot (1+\lfloor \log(B+1) \rfloor)}$, where $\tilde{\mathbf{k}} \in \{0, 1\}^{d \cdot (1+\lfloor \log(B+1) \rfloor)}$ is the binary decomposition of $\mathbf{k}$, and also prove knowledge of the vector $\mathbf{w} \in \{0, 1\}^{B+d \cdot (1+\lfloor \log(B+1) \rfloor)}$ satisfying the subset–sum relation

$$\mathbf{s} = \left[\mathbf{A} \mid -q \cdot (\mathbf{I}_d \otimes \mathbf{g}_{1+\lfloor \log(B+1) \rfloor}^\top) \right] \cdot \underbrace{\begin{bmatrix} \mathbf{b} \\ \tilde{\mathbf{k}} \end{bmatrix}}_{\triangleq \mathbf{w}} \bmod r \tag{37}$$

over $\mathbb{Z}/(r)$.

Recall that the goal is to prove knowledge of a binary $\mathbf{b} \in \{0, 1\}^B$ satisfying $\mathbf{s} = \mathbf{A} \cdot \mathbf{b} \bmod q$, for a public matrix $\mathbf{A} \in (\mathbb{Z}/(q))^{d \times B}$ and vector $\mathbf{s} \in (\mathbb{Z}/(q))^d$. Here, each element of $\mathbb{Z}/(q)$ is interpreted as an integer in the interval $[-q/2, q/2)$.

In order to be able to prove instances of variable size with a fixed common reference string, we prepare a CRS (see Figure 88) according to maximal dimensions $\bar{B}$ and $\bar{d}$ for the matrix $\mathbf{A}$. We thus define a bound $\bar{B}_1 = \bar{B} + \bar{d} \cdot (1 + \lfloor \log(\bar{B} + 1) \rfloor)$ as the maximal dimension of committed binary vectors and prove knowledge of a witness $\tilde{\mathbf{w}} = (\mathbf{b}^\top \mid \tilde{\mathbf{k}}^\top \mid \mathbf{0})^\top \in \{0, 1\}^{\bar{B}_1}$ satisfying

$$\mathbf{s} = \underbrace{\left[\mathbf{A} \mid -q \cdot (\mathbf{I}_d \otimes \mathbf{g}_{1+\lfloor \log(B+1) \rfloor}^\top) \mid \mathbf{0}^{d \times (\bar{B}_1 - (B+d \cdot (1+\lfloor \log(B+1) \rfloor)))} \right]}_{\triangleq \mathbf{A}_1 \in \mathbb{Z}^{d \times \bar{B}_1}} \cdot \underbrace{\begin{bmatrix} \mathbf{b} \\ \tilde{\mathbf{k}} \\ \mathbf{0} \end{bmatrix}}_{\triangleq \tilde{\mathbf{w}}} \bmod r \tag{38}$$

7.6.3.4   Method 2 Proof Construction: This method still proves a linear relation but it is no longer a subset sum instance since the witness is not completely binary. Specifically, it proves knowledge of a witness $\mathbf{b} = (\mathbf{b}_l^\top \mid \mathbf{b}_r^\top)^\top \in \{0, 1\}^{B_l} \times [-U_\infty, U_\infty]^{B_r}$ such that

$$\mathbf{s} = \mathbf{A} \cdot \mathbf{b} \pmod{q} \tag{39}$$

The prover and verification algorithms are given in Figure 92–Figure 96.

VC-Prove-2(p, (A, s), b):
    Where **p** is the CRS, $\mathbf{s} \in (\mathbb{Z}/(q))^d$, $\mathbf{A} = [\mathbf{A}_l \mid \mathbf{A}_r] \in (\mathbb{Z}/(q))^{d \times (B_l + B_r)}$ and q, d, B_l, B_r, U_∞ form the public statement **x**, and $\mathbf{b} = (\mathbf{b}_l \mid \mathbf{b}_r) \in \{0, 1\}^{B_l} \times [-U_\infty, U_\infty]^{B_r}$ is the private witness. Where $q \leq \bar{q}$, $d \leq \bar{d}$, and $B_l \leq \bar{B}_l$, $B_r \leq \bar{B}_r$, $U_\infty \leq \bar{U}_\infty$ for $\bar{q}, \bar{d}$ and $(\bar{U}_\infty, \bar{B}_l, \bar{B}_r)$ being in the CRS **p** which also specifies $\bar{U}_2 \triangleq \bar{U}_\infty \cdot \sqrt{\bar{B}_r}$, $\bar{U}_k \triangleq (\bar{B}_l + \bar{B}_r)/2$, $\bar{U}_H \triangleq 13.32 \cdot \sqrt{\bar{U}_2^2 + \bar{U}_k^2 \cdot \bar{d}}$ and $\bar{m} = \lceil 1 + \log \bar{U}_H \rceil$. For the proof we think of **s** and **A** consisting of elements in $\mathbb{Z}/(r)$.

1. Define $U_2 = U_\infty \cdot \sqrt{\bar{B}_r} \leq \bar{U}_2$ and $U_k = B_l/2 + B_r \cdot U_\infty/q \leq \bar{U}_k$. Let $U_H = 13.32 \cdot \sqrt{U_2^2 + U_k^2 \cdot d} \leq \bar{U}_H$ and $m = \lceil 1 + \log U_H \rceil \leq \bar{m}$.

2. Compute $\mathbf{k} \in \mathbb{Z}^d$ such that $\|\mathbf{k}\|_\infty \leq U_k$ and

$$\mathbf{s} = \mathbf{A}_l \cdot \mathbf{b}_l + \mathbf{A}_r \cdot \mathbf{b}_r - \mathbf{k} \cdot q \qquad \text{(over } \mathbb{Z})$$

3. Choose $\hat{\gamma}_e, \gamma_e \leftarrow \mathbb{Z}/(r)$ and commit to $\bar{\mathbf{b}}_r = (\mathbf{b}_r \mid (v_1, \ldots, v_4)) \in \mathbb{Z}^{B_r + 4}$ by computing

$$\hat{C}_e = \hat{g}^{\hat{\gamma}_e} \cdot \prod_{i=1}^{B_r + 4} \hat{g}_i^{\bar{\mathbf{b}}_r[i]}, \qquad C_e = g^{\gamma_e} \cdot \prod_{i=1}^{B_r + 4} g_{\bar{B}_1 + 1 - i}^{\bar{\mathbf{b}}_r[i]},$$

    where $v_1, \ldots, v_4 \in [0, U_2]$ are integers such that $\sum_{i=1}^{4} v_i^2 = U_2^2 - \|\mathbf{b}_r\|_2^2$. Also, choose $\gamma_k \leftarrow \mathbb{Z}/(r)$ and compute

$$C_k = g^{\gamma_k} \cdot \prod_{i=1}^{d} g_i^{\mathbf{k}[i]}.$$

4. Compute $\bar{\mathbf{R}} = \mathcal{H}_R(\mathbf{x}, \hat{C}_e, C_e, C_k) \in \{-1, 0, 1\}^{128 \times (\bar{B}_r + \bar{d} + 4)}$ whose columns are distributed as per $\mathcal{D}^{128}$. Let $\mathbf{R} \in \{-1, 0, 1\}^{128 \times (\bar{B}_r + d + 4)}$ the submatrix consisting of the first $B_r + d + 4$ columns of $\bar{\mathbf{R}}$. Let

$$\mathbf{w}_R = \mathbf{R} \cdot \begin{bmatrix} \bar{\mathbf{b}}_r \\ \mathbf{k} \end{bmatrix} \in \mathbb{Z}^{128}$$

    the compressed version of $\bar{\mathbf{b}}_r \in \mathbb{Z}^{d+4}$ and $\mathbf{k} \in \mathbb{Z}^d$. If $\|\mathbf{w}_R\|_\infty > U_H$, abort and return $\bot$.

5. Choose $\gamma_R \leftarrow \mathbb{Z}/(r)$ and compute

$$C_R = g^{\gamma_R} \cdot \prod_{i=1}^{128} g_i^{\mathbf{w}_R[i]}$$

    Then, compute $\vec{\phi} = (\phi_1, \ldots, \phi_{128}) = \mathcal{H}_\phi(\mathbf{x}, \mathbf{R}, \hat{C}_e, C_e, C_R, C_k) \in (\mathbb{Z}/(r))^{128}$.

6. Let $\mathbf{w}_{R,bin} \in \{0, 1\}^{128 \cdot m}$ the binary decomposition of $\mathbf{w}_R \in [-U_H, U_H]^{128}$ such that

$$\mathbf{w}_{R,bin}[(i-1) \cdot m + 1, i \cdot m] = \bar{g}_m^{-1}(\mathbf{w}_R[i]) \qquad \forall i \in [128].$$

    Define the vector $\mathbf{w}_{bin} = (\mathbf{b}_l \mid \mathbf{w}_{R,bin} \mid \mathbf{0}^{\bar{B}_1 - (B_l + 128 \cdot m)}) \in \{0, 1\}^{\bar{B}_1}$. Commit to it by computing

$$\hat{C}_{bin} = \hat{g}^{\gamma_{bin}} \cdot \prod_{i=1}^{B_l + 128 \cdot m} \hat{g}_i^{\mathbf{w}_{bin}[i]}$$

    where $\gamma_{bin} \leftarrow \mathbb{Z}/(r)$.

Figure 92: The Second Vector–Commitment–Based NIZK Prover – Part I.

Since the RLWE modulus q is much smaller than the order r of the discrete–log–hard group, we will proceed again as in [dPLS19] by proving (39) over the integers. Namely, we write $\mathbf{A} = [\mathbf{A}_l \mid \mathbf{A}_r] \in (\mathbb{Z}/(q))^{d \times (B_l + B_r)}$ and prove knowledge of $\mathbf{b}_l \in \{0, 1\}^{B_l}$, $\mathbf{b}_r \in [-U_\infty, U_\infty]^{B_r}$, and $\mathbf{k} \in \mathbb{Z}^d$ satisfying the linear relation

$$\mathbf{s} = \mathbf{A}_l \cdot \mathbf{b}_l + \mathbf{A}_r \cdot \mathbf{b}_r - \mathbf{k} \cdot q \tag{44}$$

VC-Prove-2(**p**, (**A**, **s**), **b**): (continued)
  7. Compute $\vec{\xi} = \mathcal{H}_\xi(\mathbf{x}, \hat{C}_e, C_e, \mathbf{R}, \vec{\phi}, C_R, \hat{C}_{bin}, C_k) \in (\mathbb{Z}/(r))^{128}$.
  8. Compute $\vec{y} = (y_1, \ldots, y_{\bar{B}_1}) = \mathcal{H}_{vc}(\mathbf{x}, \mathbf{R}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, C_R, \hat{C}_{bin}, C_k) \in (\mathbb{Z}/(r))^{\bar{B}_1}$. For each $i \in [B_l + 128 \cdot m + 1, \bar{B}_1]$, set $y_i = 0$. Next, choose $\gamma_y \leftarrow \mathbb{Z}/(r)$ and compute

$$C_y = g^{\gamma_y} \cdot \prod_{j=1}^{B_l + 128 \cdot m} g_{n+1-j}^{y_j \cdot \mathbf{w}_{bin}[j]}$$

Compute $\vec{t} = (t_1, \ldots, t_{\bar{B}_1}) = \mathcal{H}_t(\mathbf{x}, \vec{y}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.
  9. Compute $\vec{\theta} = \mathcal{H}_{linmap}(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y) \in (\mathbb{Z}/(r))^{\bar{d}+1}$ and let the matrix $\tilde{\mathbf{A}} = [\mathbf{A}_l \mid \mathbf{A}_r \mid -q \cdot \mathbf{I}_d] \in \mathbb{Z}^{d \times (B_l + B_r + d))}$ from (45). Let $\vec{\theta}_0 \in (\mathbb{Z}/(r))^d$ the first d entries of $\vec{\theta}$.
  10. Let $t_\theta = \vec{\theta}_0^\top \cdot \mathbf{s} \bmod r, \vec{a}_{\theta,l}^\top = \vec{\theta}_0^\top \cdot \mathbf{A}_l \in \mathbb{Z}^{1 \times B_l} \bmod r$ and $\vec{a}_{\theta,r}^\top = \vec{\theta}_0^\top \cdot \mathbf{A}_r \in \mathbb{Z}^{1 \times B_r} \bmod r$. Let $\bar{\vec{a}}_{\theta,l}^\top = (\vec{a}_{\theta,l}^\top \mid \mathbf{0}^{\bar{B}_1 - B_l}), \bar{\vec{a}}_{\theta,r}^\top = (\vec{a}_{\theta,r}^\top \mid \mathbf{0}^{\bar{B}_1 - B_r}), \bar{\vec{\theta}}_0 = (\vec{\theta}_0 \mid \mathbf{0}^{\bar{B}_1 - d}) \in (\mathbb{Z}/(r))^{\bar{B}_1}$.
  11. Compute $\vec{\omega} = (\omega_1, \ldots, \omega_{\bar{B}_1}) = \mathcal{H}_\omega(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y)$.
  12. Let $\vec{\delta} = (\delta_r, \delta_{dec}, \delta_{eq}, \delta_y, \delta_\theta, \delta_e, \delta_\ell) = \mathcal{H}_{agg'}(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\omega}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y)$.
  13. Let $\bar{\bar{\theta}}_0 = (\vec{\theta}_0 \mid \mathbf{0}^{\bar{B}_1 - d}) \in (\mathbb{Z}/(r))^{\bar{B}_1}$ and define matrices $\mathbf{R}' \in (\mathbb{Z}/(r))^{128 \times \bar{B}_1}$ and $\mathbf{H}_\xi \in (\mathbb{Z}/(r))^{128 \times (B_l + 128 \cdot m)}$ such that

$$\forall i \in [128] : \mathbf{R}'[i,j] = \begin{cases} \mathbf{R}[i,j] & \text{if } j \in [1, B_r + d + 4] \\ 0 & \text{if } j \in [B_r + d + 5, \bar{B}_1]. \end{cases}$$

$$\forall i \in [128] : \mathbf{H}_\xi[i,j] = \begin{cases} \vec{\xi}[i] \cdot \mathbf{H}[i, j - B_l] & \text{if } j \in [B_l + 1, B_l + 128 \cdot m] \\ 0 & \text{if } j \in [1, B_l]. \end{cases}$$

Figure 93: The Second Vector–Commitment–Based NIZK Prover – Part II.

over the integers. As in [dPLS19], the latter equality is proven over $\mathbb{Z}$ by providing evidence that

$$\mathbf{s} = \underbrace{\left[\begin{array}{ccc} \mathbf{A}_l & \mathbf{A}_r & -q \cdot \mathbf{I}_d \end{array} \right]}_{\triangleq \tilde{\mathbf{A}}} \cdot \begin{bmatrix} \mathbf{b}_l \\ \mathbf{b}_r \\ \mathbf{k} \end{bmatrix} \bmod r, \tag{45}$$

where r is the order of the discrete–log–hard group. As long (45) holds for an extended witness $(\mathbf{b}_l \mid \mathbf{b}_r \mid \mathbf{k})$ of sufficiently small infinity norm, both members are guaranteed to have infinity norm smaller than $r/2$, which implies that (45) holds over $\mathbb{Z}$ and not only modulo r. In (44), we note that $\|\mathbf{k}\|_\infty \leq B_l/2 + B_r U_\infty$. If the prover can convince the verifier that $\|\mathbf{k}\|_\infty \leq B_l/2 + B_r \cdot U_\infty, \|\mathbf{b}_r\| \leq U_\infty$ and $\mathbf{b}_l \in \{0, 1\}^{B_l}$, the verifier knows that the right–hand–side member of (45) is smaller than $B_l \cdot q + 2B_r \cdot U_\infty$ in infinity norm. Hence, if $r/2 > B_l \cdot q + 2B_r \cdot U_\infty$, the verifier is convinced that (44) holds over $\mathbb{Z}$. In order to prove the smallness of $(\mathbf{b}_l \mid \mathbf{b}_r \mid \mathbf{k})$, we do not directly decompose $\mathbf{b}_r$ and $\mathbf{k}$ into bits because it would significantly increase the dimension of committed vectors in the vector commitment scheme. Instead, we first project $(\mathbf{b}_r, \mathbf{k})$ to a smaller dimension 128 by computing $\mathbf{w}_R = \mathbf{R} \cdot (\mathbf{b}_r^\top \mid \mathbf{k}^\top)^\top \bmod r$ for a matrix $\mathbf{R} \in \{-1, 0, 1\}^{128 \times (B_r + d)}$ sampled from a distribution $\mathcal{D}$ that outputs zero with probability $1/2$ and one and -1 with probability $1/4$ each. We then prove that $\mathbf{w}_R \in \mathbb{Z}^{128}$ has small infinity norm via its binary decomposition, which is much cheaper than directly proving the smallness since the dimension of $\mathbf{w}_R$ is only 128 (instead of 2048 or more).

VC-Prove-2$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$: (continued)

14. Compute the polynomial

$$P_\pi[X] = \Big(\delta_y \cdot \gamma_y +$$

$$\Big(\sum_{j=1}^{B_l+128m} \big(\delta_y \cdot y_j \cdot (\mathbf{w}_{bin}[j] - 1) + \delta_\theta \cdot \bar{\bar{a}}_{\theta,l}[j] + \delta_{eq} \cdot t_j \cdot y_j$$

$$+ \delta_{dec} \cdot \sum_{i=1}^{128} \mathbf{H}_\xi[i,j]\big) \cdot X^{\bar{B}_1+1-j}\Big)$$

$$\Big) \cdot \Big(\gamma_{bin} + \sum_{j=1}^{B_l+128m} \mathbf{w}_{bin}[j] \cdot X^j\Big)$$

$$+ \Big(\delta_\ell \cdot \big(\gamma_e + \sum_{j=1}^{B_r+4} \bar{e}[j] \cdot X^{\bar{B}_1+1-j}\big)$$

$$+ \sum_{j=1}^{\bar{B}_1} \big(\delta_\theta \cdot \bar{\bar{a}}_{\theta,r}[j] + \sum_{i=1}^{128} \delta_r \cdot \phi_i \cdot \mathbf{R}'[i,j] + \delta_e \cdot \omega_j\big) \cdot X^{\bar{B}_1+1-j}$$

$$\Big) \cdot \big(\hat{\gamma}_e + \sum_{j=1}^{B_r+4} \bar{e}[j] \cdot X^j\big)$$

$$+ \Big(\gamma_r + \sum_{j=1}^{d} \mathbf{k}[j] \cdot X^j\Big) \cdot \Big(\sum_{j=1}^{d} \big(-\delta_\theta \cdot q \cdot \bar{\theta}_0[j] + \sum_{i=1}^{128} \delta_r \cdot \phi_i \cdot \mathbf{R}[i, B_r + 4 + j]\big) \cdot X^{\bar{B}_1+1-j}\Big)$$

$$- \Big(\gamma_R + \sum_{j=1}^{128} \mathbf{w}_R[j] \cdot X^j\Big) \cdot \Big(\sum_{j=1}^{128} (\delta_r \cdot \phi_j + \delta_{dec} \cdot \bar{\xi}[j]) \cdot X^{\bar{B}_1+1-j}\Big)$$

$$- \delta_e \cdot \big(\gamma_e + \sum_{j=1}^{B_r+4} \bar{e}[j] \cdot X^{\bar{B}_1+1-j}\big) \cdot \Big(\sum_{j=1}^{B_r+4} \omega_j \cdot X^j\Big)$$

$$- \delta_{eq} \cdot \Big(\gamma_y + \sum_{j=1}^{B_l+128m} y_j \cdot \mathbf{w}_{bin}[j] \cdot X^{\bar{B}_1+1-j}\Big) \cdot \Big(\sum_{j=1}^{\bar{B}_1} t_j \cdot X^j\Big) - (\delta_\theta \cdot t_\theta + \delta_\ell \cdot U_2^2) \cdot X^{\bar{B}_1+1}.$$

From the coefficients of $P_\pi[X] = \sum_{i=1}^{\bar{B}_1+B_l+128m} v_i \cdot X^i$, compute $\pi = g^{P_\pi(\alpha)} = g^{v_0} \cdot \prod_{i=1, i \neq \bar{B}_1+1}^{\bar{B}_1+B_l+128m} g_i^{v_i}$.

Figure 94: The Second Vector–Commitment–Based NIZK Prover – Part III.

VC-Prove-2(**p**, (**A**, **s**), **b**): (continued)

15. Compute deterministic KZG commitments $\hat{C}_t = \prod_{j=1}^{\bar{B}_1} \hat{g}_j^{t_j}$, $\hat{C}_\omega = \prod_{i=1}^{B_r+4} \hat{g}_i^{\omega_i}$ and

$$C_{h,1} = \prod_{j=1}^{B_l+128 \cdot m} g_{\bar{B}_1+1-j}^{\delta_\theta \cdot \bar{\bar{a}}_\theta[j] - \delta_y \cdot y_j + \delta_{eq} \cdot t_j \cdot y_j + \delta_{dec} \cdot \sum_{i=1}^{128} \mathbf{H}_\xi[i,j]}$$

$$C_{h,2} = \prod_{j=1}^{\bar{B}_1} g_{\bar{B}_1+1-j}^{\delta_\theta \cdot \bar{\bar{a}}_\theta[j] + \sum_{i=1}^{128} \delta_r \cdot \phi_i \cdot \mathbf{R}'[i,j] + \delta_e \cdot \omega_j}$$

$$\hat{C}_{h,3} = \prod_{j=1}^{d} \hat{g}_{\bar{B}_1+1-j}^{-\delta_\theta \cdot q \cdot \vec{\theta}_0[j] + \delta_r \cdot \sum_{i=1}^{128} \phi_i \cdot \mathbf{R}[i, B_r+4+j]}$$

to the polynomials $P_t[X] = \sum_{i=1}^{\bar{B}_1} t_i \cdot X^i, P_\omega[X] = \sum_{i=1}^{B_r+4} \omega_i \cdot X^i$ and

$$P_{h,1}[X] = \sum_{j=1}^{B_l+128m} \left(\delta_\theta \cdot \bar{a}_{\theta,l}[j] - \delta_y \cdot y_j + \delta_{eq} \cdot t_j \cdot y_j + \delta_{dec} \cdot \sum_{i=1}^{128} \mathbf{H}_\xi[i,j] \right) \cdot X^{\bar{B}_1+1-i}$$

$$P_{h,2}[X] = \sum_{j=1}^{\bar{B}_1} \left(\delta_\theta \cdot \bar{a}_{\theta,l}[j] + \sum_{i=1}^{128} \delta_r \cdot \phi_i \cdot \mathbf{R}'[i,j] + \delta_e \cdot \omega_j \right) \cdot X^{\bar{B}_1+1-j}$$

$$P_{h,3}[X] = \sum_{j=1}^{d} \left(-\delta_\theta \cdot q \cdot \vec{\theta}_0[j] + \delta_r \cdot \sum_{i=1}^{128} \phi_i \cdot \mathbf{R}[i, B_r + 4 + j] \right) \cdot X^{\bar{B}_1+1-j} \tag{40}$$

16. Compute $z = \mathcal{H}_z\big(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\delta}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega\big) \in \mathbb{Z}/(r)$ as an evaluation input for the polynomials (40). Let the evaluations

$$(p_{h,1}, p_{h,2}, p_t, p_{h,3}, p_\omega) = (P_{h,1}(z), P_{h,2}(z), P_t(z), P_{h,3}(z), P_\omega(z)). \tag{41}$$

17. Compute a batch evaluation proof for the polynomial evaluations (41). To do this, com–pute

$$\chi = \mathcal{H}_\chi\big(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\delta}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega,$$
$$z, p_{h,1}, p_{h,2}, p_t, p_{h,3}, p_\omega\big).$$

Then compute the polynomial $Q[X] = \sum_{i=0}^{\bar{B}_1-1} q_i \cdot X^i$ such that

$$Q_{KZG}[X] = \big(P_{h,1}[X] + \chi \cdot P_{h,2}[X] + \chi^2 \cdot P_{h,3}[X] + \chi^3 \cdot P_t[X] + \chi^4 \cdot P_\omega[X]\big)$$
$$- (p_{h,1} + p_{h,2} \cdot \chi + \chi^2 \cdot p_{h,3} + \chi^3 \cdot p_t + \chi^4 \cdot p_\omega)\big)/(X - z)$$

Finally, compute $\pi_{KZG} = g^{q_0} \cdot \prod_{i=1}^{\bar{B}_1-1} g_i^{q_i} = g^{Q_{KZG}(\alpha)}$.

18. Output the final proof

$$prf = \big(\hat{C}_e, C_e, C_k, C_R, \hat{C}_{bin}, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega, \pi, \pi_{KZG}\big).$$

Figure 95: The Second Vector–Commitment–Based NIZK Prover – Part IV.

VC-Verify-2$(\mathbf{p}, (\mathbf{A}, \mathbf{s}, U_\infty), prf)$:

Given a statement $(\mathbf{A}, \mathbf{s}, U_\infty)$ which, along with q, d, B_l, B_r form the public statement $\mathbf{x}$ and a purported proof

$$prf = \big(\hat{C}_e, C_e, C_k, C_R, \hat{C}_{bin}, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega, \pi, \pi_{KZG}\big) \in \hat{\mathbb{G}} \times \mathbb{G}^3 \times \hat{\mathbb{G}} \times \mathbb{G}^3 \times \hat{\mathbb{G}}^3 \times \mathbb{G}^2,$$

return zero if prf does not parse properly. Otherwise, set $U_2 = U_\infty \cdot \sqrt{B_r}$ and do the following.

1. Compute $\bar{\mathbf{R}} = \mathcal{H}_R(\mathbf{x}, \hat{C}_e, C_e, C_k) \in \{-1, 0, 1\}^{128 \times (\bar{B}_r + \bar{d} + 4)}$ whose columns are distributed as per $\mathcal{D}^{128}$. Let $\mathbf{R} \in \{-1, 0, 1\}^{128 \times (d+4)}$ the submatrix consisting of the first d columns of $\bar{\mathbf{R}}$.

2. Set $\vec{\phi} = \mathcal{H}_\phi(\mathbf{x}, \mathbf{R}, \hat{C}_e, C_e, C_R, C_k) \in (\mathbb{Z}/(r))^{128}$, $\vec{\xi} = \mathcal{H}_\xi(\mathbf{x}, \hat{C}_e, C_e, \mathbf{R}, \vec{\phi}, C_R, \hat{C}_{bin}, C_k) \in (\mathbb{Z}/(r))^{128}$, and $\vec{y} = \mathcal{H}_{vc}(\mathbf{x}, \mathbf{R}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, C_R, \hat{C}_{bin}, C_k) \in (\mathbb{Z}/(r))^{\bar{B}_1}$. For each index $i \in [B_l + 128 \cdot m + 1, \bar{B}_1]$, set $y_i = 0$. Then, compute

    $$\vec{t} = \mathcal{H}_t(\mathbf{x}, \vec{y}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y) \in (\mathbb{Z}/(r))^{\bar{B}_1},$$

    $$\vec{\theta} = \mathcal{H}_{linmap}(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y) \in (\mathbb{Z}/(r))^{\bar{d}+1}$$

    $$\vec{\omega} = \mathcal{H}_\omega(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \hat{C}_e, C_e, C_R, \hat{C}_{bin}, C_k, C_y) \in (\mathbb{Z}/(r))^{\bar{B}_1},$$

    $$\vec{\delta} = (\delta_r, \delta_{dec}, \delta_{eq}, \delta_y, \delta_\theta, \delta_e, \delta_\ell) = \mathcal{H}_{agg'}(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\omega}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y)$$

3. Let the matrix $\tilde{\mathbf{A}} = [\mathbf{A}_l \mid \mathbf{A}_r \mid -q \cdot \mathbf{I}_d] \in \mathbb{Z}^{d \times (B_l + B_r + d)}$ from (45). Let $\vec{\theta}_0 \in (\mathbb{Z}/(r))^d$ the first d entries of $\vec{\theta}$. Let $t_\theta = \vec{\theta}_0^\top \cdot \mathbf{s} \bmod r$, $\vec{a}_{\theta,l}^\top = \vec{\theta}_0^\top \cdot \mathbf{A}_l \in \mathbb{Z}^{1 \times B_l} \bmod r$ and $\vec{a}_{\theta,r}^\top = \vec{\theta}_0^\top \cdot \mathbf{A}_r \in \mathbb{Z}^{1 \times B_r} \bmod r$.

4. Let $\bar{\vec{\theta}}_0 = (\vec{\theta}_0 \mid \mathbf{0}^{n-(d+k)}) \in (\mathbb{Z}/(r))^{\bar{B}_1}$ and define the matrices $\mathbf{R}' \in (\mathbb{Z}/(r))^{128 \times \bar{B}_1}$ and $\mathbf{H}_\xi \in (\mathbb{Z}/(r))^{128 \times (B_l + 128 \cdot m)}$ as in step 13 of *VC-Prove-2*. Return zero if the following equality does not hold:

    $$e\Big(C_y^{\delta_y} \cdot C_{h,1}, \ \hat{C}_{bin}\Big) \cdot e\Big(C_e^{\delta_\ell} \cdot C_{h,2}, \hat{C}_e\Big) \cdot e\big(C_{\tilde{r}}, \hat{C}_{h,3}\big) \cdot e\Big(C_R, \prod_{j=1}^{128} \hat{g}_{n+1-j}^{\delta_r \cdot \phi_j + \delta_{dec} \cdot \vec{\xi}[j]}\Big)^{-1}$$

    $$\cdot e(C_e^{\delta_e}, \hat{C}_\omega)^{-1} \cdot e(C_y^{\delta_{eq}}, \hat{C}_t)^{-1} \cdot e(g_1, \hat{g}_{\bar{B}_1})^{-\delta_\theta \cdot t_\theta - \delta_\ell \cdot U_2^2} = e(\pi, \hat{g}) \qquad (42)$$

5. Compute

    $$z = \mathcal{H}_z(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\delta}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega)$$

    and

    $$\chi = \mathcal{H}_\chi(\mathbf{x}, \vec{y}, \vec{t}, \vec{\phi}, \vec{\xi}, \vec{\theta}, \vec{\delta}, \hat{C}_e, C_e, \mathbf{R}, C_R, \hat{C}_{bin}, C_k, C_y, C_{h,1}, C_{h,2}, \hat{C}_t, \hat{C}_{h,3}, \hat{C}_\omega,$$
    $$z, p_{h,1}, p_{h,2}, p_t, p_{h,3}, p_\omega).$$

    Compute $(p_{h,1}, p_{h,2}, p_{h,3}, p_t, p_\omega) = (P_{h,1}(z), P_{h,2}(z), P_{h,3}(z), P_t(z), P_\omega(z))$ by evaluating the polynomials in (40).

6. Return zero if the following equality does not hold:

    $$e\big(C_{h,1} \cdot C_{h,2}^\chi \cdot g^{-p_{h,1} - \chi \cdot p_{h,2}}, \hat{g}\big)$$
    $$\cdot e\big(g, \hat{C}_{h,3}^{(\chi^2)} \cdot \hat{C}_t^{(\chi^3)} \cdot \hat{C}_\omega^{(\chi^4)} \cdot \hat{g}^{-(\chi^2) \cdot p_{h,3} - (\chi^3) \cdot p_t - (\chi^4) \cdot p_\omega}\big) = e\big(\pi_{KZG}, \hat{g}_1 \cdot \hat{g}^{-z}\big). \qquad (43)$$

If equalities (42) and (43) both hold, return one.

Figure 96: The Second Vector–Commitment–Based NIZK Verifier.

7.6.4 ZKPoKs Based on MPC–in–the–Head

The basic MPC–in–the–Head protocols are parameterized by a number of hyper-parameters;
- n: The number of parties in the MPC–in–the–Head protocol.
- M: The total number of executions of the MPC protocol which are committed to.
- τ: The total number of executions which are opened in the online phase.

We present three different ZKPoKs based on MPC in the head. One for a general value of q, based on full threshold sharing (from [FMRV22]) a second one which is an optimization of the first technique also for general q (from [AGH+23]), and a third for $q = p_1 \cdots p_k$, for large primes p_i, which is an adaption of the method from [FR23].

In the ZKPoKs based on full threshold secret sharing (from [FMRV22]) we have the additional parameters:
- η: The number of opening rejections allowed.
- A: A parameter used to "squish" the commitments down (see [FMRV22]), we assume $A = 2^a$ in what follows.

In the ZKPoKs based on full threshold sharing, which make use of the hyper–cube trick from [AGH+23], we have the additional two parameters related by $n = N^D$.
- N: The length of the hyper–cube edge.
- D: The hyper–cube dimension.

In the ZKPoKs based on Shamir sharing (from [FR23]) we have the additional parameters:
- t: The threshold for the underlying Shamir scheme.

All proofs require access to some XOF–like object as per Section 5.6.3, and a *TreePRG* algorithm, as per Section 7.1.6 for a security parameter *sec*. In addition, the ZKPoKs can easily be changed into a signature proof of knowledge, by simply hashing in any desired message into the hash function invocations of $\mathcal{H}_2$ and $\mathcal{H}_4$ below.

7.6.4.1 Method 1: Full Threshold Based Proofs: In this section we let $[x]$ denote an additive sharing over $\mathbb{Z}/(q)$ of a value $x \in \mathbb{Z}/(q)$, i.e.

$$x = x_1 + \cdots + x_n \pmod{q}.$$

with $x_i \in \mathbb{Z}/(q)$. Note the number of players n here is different from that considered earlier, the precise number is a parameter of our zero-knowledge proof. We et $[x]^{(i)}$ denote the value x_i in the sharing, i.e. the i–th players share.

We let $\mathcal{H}_1, \ldots, \mathcal{H}_6$ denote six independent collision resistant hash functions. The first four hash functions $\mathcal{H}_1, \ldots, \mathcal{H}_4$ are normal hash functions, with normal bit–string outputs. We define these as follows:

$$\mathcal{H}_1(x) = Hash^{2 \cdot sec}(DSep(KKW1)\|x),$$
$$\mathcal{H}_2(x) = Hash^{2 \cdot sec}(DSep(KKW2)\|x),$$
$$\mathcal{H}_3(x) = Hash^{2 \cdot sec}(DSep(KKW3)\|x),$$
$$\mathcal{H}_4(x) = Hash^{2 \cdot sec}(DSep(KKW4)\|x).$$

However $\mathcal{H}_5$ and $\mathcal{H}_6$ are special.
- The function $\mathcal{H}_5$ will take an input string and output a random subset $J \subset \{1, \ldots, M\}$ of size τ.
- The function $\mathcal{H}_6$ will take an input string and output a random element in $\{1, \ldots, n\}^\tau$.

The pseudo–code for these two functions are given in Figure 97. The proofs are presented in Figure 98– Figure 100.

> **Hash Functions $\mathcal{H}_5$ and $\mathcal{H}_6$**
>
> *Shuffle*(*XOF*, $A_1, \ldots, A_n$):
>   This implements a Fisher–Yates shuffle of n elements based on the randomness provided by
>   the *XOF*. Note that n is exponentially smaller than 2^{64}.
>   1. For $i \in [n, 1]$:
>   (a) $r \leftarrow XOF.Next(64)$, treat this as an integer over 64 bits.
>   (b) $k \leftarrow (r \bmod i) + 1$.
>   (c) *Swap*(A_i, A_k).
>   i.e. execute $t \leftarrow A_k, A_k \leftarrow A_i, A_i \leftarrow t$.
>   2. Return A
>
> $\mathcal{H}_5(x)$:
>   Hash x into a random subset of $\{1, \ldots, M\}$ of size τ.
>   1. $y \leftarrow Hash^{2 \cdot sec}(DSep(KKW5)\|x)$.
>   2. $XOF.Init(y, DSep(KKW51))$
>   3. For $i \in [1, M]$:
>   (a) $A_i \leftarrow i$.
>   4. $B \leftarrow Shuffle(XOF, A)$.
>   5. Return $B_1, \ldots, B_\tau$
>
> *RandElem*(*XOF*, n, τ):
>   Obtain a random element of $\{1, \ldots, n\}^\tau$, assumes $n \ll 2^{64}$.
>   1. $y \leftarrow Hash^{2 \cdot sec}(DSep(KKW6)\|x)$.
>   2. $XOF.Init(y, DSep(KKW61))$
>   3. For $i \in [1, \tau]$
>   (a) $A_i \leftarrow (XOF.Next(64) \bmod n) + 1$.
>   4. Return $(A_1, \ldots, A_\tau)$.
>
> $\mathcal{H}_6(x)$:
>   Hash x into a random element of $\{1, \ldots, n\}^\tau$
>   1. $y \leftarrow Hash^{2 \cdot sec}(DSep(KKW6)\|x)$.
>   2. $XOF.Init(y, DSep(KKW61))$.
>   3. $(A_1, \ldots, A_\tau) \leftarrow RandElem(XOF, n, \tau)$.
>   4. Return $(A_1, \ldots, A_\tau)$.

Figure 97: $\mathcal{H}_5$ and $\mathcal{H}_6$.

KKW Protocol for Multiple Subset Sum Equations: Part I

MPCitHead-Prove-1$((\mathbf{A}, \mathbf{s}), \mathbf{b})$:
    Where $\mathbf{s} \in (\mathbb{Z}/(q))^d$ and $\mathbf{A} \in (\mathbb{Z}/(q))^{d \times B}$ are public, and $\mathbf{b} \in \{0, 1\}^B$ is private.

Phase 1: First Commitment
1. $seed \leftarrow \{0, 1\}^{sec}$.
2. $\{seed_i\}_{i=1}^M \leftarrow$ *TreePRG.Gen*$(seed, \lceil \log_2 M \rceil)$.
3. For $e \in [1, \ldots, M]$:
    (a) $\{seed_{e,i}\}_{i=1}^n \leftarrow$ *TreePRG.Gen*$(seed_e, \lceil \log_2 n \rceil)$.
    (b) *XOF.Init*$(seed_e, DSep(KKW1))$, add the string for domain separation with the *TreePRG*.
    (c) $\mathbf{r}_e \leftarrow$ *XOF.Next*(B). These are the input masks for the bits, we think of $\mathbf{r}_e \in \{0, 1\}^B$.
    (d) For $i \in [1, \ldots, n]$:
        i. $\rho_{e,i} \leftarrow$ *XOF.Next*(sec).
        ii. $comm_{e,i} \leftarrow$ *Commit*$(seed_{e,i}; \rho_{e,i})$.
        iii. $XOF'.Init(seed_{e,i})$.
        iv. For $\ell \in [1, \ldots, B]$:
            A. $[r_{e,\ell}]^{(i)} \leftarrow XOF'.Next(a)$, treating the result as an integer in $\{0, \ldots, A - 1\}$.
    (e) For $\ell \in [1, \ldots, B]$:
        i. $\Delta_{e,\ell} \leftarrow \sum_{i=1}^n [r_{e,\ell}]^{(i)} \pmod{q} - \mathbf{r}_e^{(\ell)}$. Note that $\Delta_{e,\ell} \in [-1, \ldots, n \cdot (A - 1)]$ and thus needs at most $\log_2(n \cdot A)$ bits to represent it.
        ii. $[r'_{e,\ell}] \leftarrow [r_{e,\ell}]$.
        iii. $[r'_{e,\ell}]^{(n)} \leftarrow [r'_{e,\ell}]^{(n)} - \Delta_{e,\ell} \pmod{q}$. This means that $[r'_{e,\ell}]$ is now a sharing over $\mathbb{Z}/(q)$ of the *bit* $\mathbf{r}_e^{(\ell)}$.
    (f) $h_e \leftarrow \mathcal{H}_1\left(\{\Delta_{e,i}\}_{i=1}^B \, \| \, \{comm_{e,i}\}_{i=1}^n\right)$.
4. $h \leftarrow \mathcal{H}_2(h_1, \ldots, h_M)$.
5. $prf_1 \leftarrow h$.

Figure 98: KKW Protocol for Multiple Subset Sum Equations: Part I.

┌───┐
│ **KKW Protocol for Multiple Subset Sum Equations: Part II** │
└───┘

Phase 2: Second Commitment

1. $J \leftarrow \mathcal{H}_5(h)$.
2. For $e \in J$:
   (a) $\mathbf{y}_e \leftarrow \mathbf{b} \oplus \mathbf{r}_e$.
   (b) For $\ell \in [1, \ldots, B]$:
      i. $[b_{e,\ell}] \leftarrow [r'_{e,\ell}] + \mathbf{y}_e^{(\ell)} - 2 \cdot \mathbf{y}_e^{(\ell)} \cdot [r'_{e,\ell}]$. This linear operation on shares, makes $[b_{e,\ell}]$ a sharing of the input bit $\mathbf{b}^{(\ell)}$.
   (c) Let $\{[s_{e,j}]\}_{j=1}^d$ denote the sharing of the values in the subset–sum statement being proved, Recall this will be a linear function of the values $\{[b_{e,\ell}]\}_{\ell=1}^B$. i.e. we set, for $j = 1, \ldots, d$,
   $$[s_{e,j}] \leftarrow \sum_{\ell=1}^{B} \mathbf{A}_{j,\ell} \cdot [b_{e,\ell}].$$
   (d) $h'_e \leftarrow \mathcal{H}_3\left(\mathbf{y}_e \,\|\, \{\{[s_{e,j}]^{(i)}\}_{j=1}^d\}_{i=1}^n\right)$.
3. $h' \leftarrow \mathcal{H}_4(\{h'_e\}_{e \in J})$.
4. $D \leftarrow TreePRG.Punc(seed, \lceil \log_2 M \rceil, J)$:
5. $prf_1 \leftarrow prf_1 \,\|\, h' \,\|\, D$.

Phase 3: Completion of Proof

1. $L \leftarrow \mathcal{H}_6(prf_1)$. Write $L = \{l_e\}_{e \in J} \leftarrow \in \{1, \ldots, n\}^\tau$.
2. $cnt \leftarrow 0$.
3. If there exist values $(e, \ell) \in J \times \{1, \ldots, B\}$ such that
   · Either $\mathbf{r}_e^{(\ell)} = 1$ and $[r_{e,\ell}]^{(l_e)} = 0$,
   · Or $\mathbf{r}_e^{(\ell)} = 0$ and $[r_{e,\ell}]^{(l_e)} = A - 1$,
   then $prf \leftarrow prf_1 \,\|\, (0 \,\|\, h_e \,\|\, h'_e)$ and set $cnt \leftarrow cnt + 1$, else $prf \leftarrow prf_1 \,\|\, 1$. In the former case we say this value of e is rejected, otherwise we say it is accepted. The verifier can determine the state from the appended zero or one bit.
4. If $cnt \geq \eta$ then abort the proof and start again.
5. For $e \in J$ and e accepted:
   (a) $D_e \leftarrow TreePRG.Punc(seed_e, \lceil \log_2 n \rceil, \{l_e\})$.
   (b) $aux_e \leftarrow D_e$, the value aux_e will contain a string of values.
   (c) For $\ell \neq l_e$:
      i. $aux_e \leftarrow aux_e \,\|\, \rho_{e,\ell}$.
   (d) For $\ell \in [1, \ldots, B]$:
      i. $\tilde{r}_{e,\ell} \leftarrow \mathbf{r}_e^{(\ell)} - [r_{e,\ell}]^{(l_e)}$.
   (e) $aux_e \leftarrow aux_e \,\|\, \{\tilde{r}_{e,\ell}\}_{\ell=1}^B \,\|\, \mathbf{y}_e \,\|\, comm_{e,l_e}$.
6. $prf \leftarrow prf \,\|\, \{aux_e\}_{e \in J}$.
7. Return prf.

Figure 99: KKW Protocol for Multiple Subset Sum Equations: Part II.

KKW Protocol for Multiple Subset Sum Equations – Verification

MPCitHead-Verify-1$((\mathbf{A}, \mathbf{s}), prf)$:

1. Parse the proof prf into it's constituent parts.
2. $J \leftarrow \mathcal{H}_5(h)$.
3. $L \leftarrow \mathcal{H}_6(prf_1)$. Write $L = \{l_e\}_{e \in J}$.
4. $\{seed_e\}_{e \notin J} \leftarrow TreePRG.GenPunc(D, J, \lceil \log_2 M \rceil)$.
5. For $e \notin J$ compute h_e as in Step 3 of Figure 98 using $seed_e$.
6. For $e \in J$ and e rejected, recover h_e and h'_e from the values supplied in the proof.
7. For $e \in J$ and e accepted:
    (a) $\{seed_{e,i}\}_{i \neq l_e} \leftarrow TreePRG.GenPunc(D_e, \{l_e\}, \lceil \log_2 n \rceil)$.
    (b) For $i \neq l_e$:
        i. $comm_{e,i} \leftarrow Commit(seed_{e,i}; \rho_{e,i})$.
        ii. Compute $[r_{e,\ell}]^{(i)}$ using $seed_{e,i}$ just as in Step 3(d)iv of Figure 98.
    (c) For $\ell \in [1, \ldots, B]$:
        i. $\Delta_{e,\ell} \leftarrow \sum_{i \neq l_e} [r_{e,\ell}]^{(i)} - \tilde{r}_{e,\ell}$.
    (d) $h_e \leftarrow \mathcal{H}_1 \left(\{\Delta_{e,i}\}_{i=1}^B \| \{comm_{e,i}\}_{i=1}^n \right)$.
    (e) For $i \neq l_e$:
        i. Compute $[r'_{e,\ell}]^{(i)}$, using the $\Delta_{e,i}$ values, just as in Steps 3e of Figure 98.
        ii. Compute $[b_{e,\ell}]^{(i)}$, using $\mathbf{y}_e$, just as in Step 2b of Figure 99.
        iii. Compute $[s_{e,j}]^{(i)}$ for $j = 1, \ldots, d$ using Step 2c of Figure 99.
    (f) Compute $[s_{e,j}]^{(l_e)}$ for $j = 1, \ldots, d$ by subtracting $\sum_{i \neq l_e} [s_{e,j}]^{(i)}$ from the actual subset–sum values, i.e. compute for $j = 1, \ldots, d$,

    $$[s_{e,j}]^{(l_e)} \leftarrow s_j - \sum_{i \neq l_e} [s_{e,j}]^{(i)}.$$

    (g) $h'_e \leftarrow \mathcal{H}_3 \left(\mathbf{y}_e \| \{\{[s_{e,j}]^{(i)}\}_{j=1}^d\}_{i=1}^n \right)$.
8. If $h' \neq \mathcal{H}_4(\{h'_e\}_{e \in J})$ then reject the proof.
9. If $h \neq \mathcal{H}_2(h_1, \ldots, h_M)$ then reject the proof.

Figure 100: KKW Protocol for Multiple Subset Sum Equations – Verification.

7.6.4.2   Method 2: Full Threshold Based Proofs – Hyper–Cube Variant: This variant of the above proof is based on the strategy from [AGH+23]. The basic idea is to partition the above n players into a hyper–cube of dimension D, and side length N, so that $n = N^D$. We then in the main MPC part of the proof execute D executions of the MPC protocol amongst N parties, instead of one MPC protocol amongst N^D parties.

Given a secret sharing of $[r]$, where each player $i \in [1, \ldots, n]$ holds the value $[r]^{(i)}$, we create D different sharings of the same value amongst N different players as follows. We first map $i \in [1, \ldots, n]$ into a vector $\mathbf{i}$ representing a coordinate in the hyper–cube of size N^D, i.e. we effectively write i in base N. Write $\mathbf{i} = (i_1, \ldots, i_D)$ with $i_j \in [1, \ldots, N]$. By abuse of notation, we write $[r]^{(\mathbf{i})}$ to represent the value of $[r]^{(i)}$ under this mapping.

For each $v \in [1, \ldots, D]$ we can now define a new sharing of the value r shared in $[r]$ amongst players $k \in [1, \ldots, N]$ by setting

$$
\begin{aligned}
[[r]]^{(v,k)} &= \sum_{i_1=1}^{N} \cdots \sum_{i_{v-1}=1}^{N} \sum_{i_{v+1}=1}^{N} \cdots \sum_{i_D=1}^{N} [r]^{(i_1,\ldots,i_{v-1},k,i_{v+1},\ldots,i_D)} \\
&= \sum_{i_t \in [1,\ldots,N], t \neq j} [r]^{(i_1,\ldots,i_{v-1},j,i_{v+1},\ldots,i_D)}.
\end{aligned}
\tag{46}
$$

We then have that, for each v, the values $[[r]]^{(v,\cdot)}$ form an additive sharing of r since

$$
[[r]]^{(v,1)} + \ldots + [[r]]^{(v,N)} = r.
$$

Phase 1 of the proof is executed exactly as in Figure 98, in particular this means the correction values $\Delta_{e,i}$ are applied to the full sharing over the n–parties. This means we can apply the same abort probability analysis as for the original variant of the proof.

However, for Phase 2 we switch from the normal n party sharing into our D sharings via the hyper–cube. This is explained in Figure 101. Note that, the "cost" of executing Phase 2 goes from something proportional to

$$
d \cdot B \cdot \tau \cdot (n-1) = d \cdot B \cdot \tau \cdot (N^D - 1)
$$

to something more akin to

$$
B \cdot \left(N^D + d \cdot D \cdot (N-1) \right).
$$

Thus for "big" d this should give an improvement in the execution time.

Phase 3 is identical to that given in Figure 99. In particular it is the tweak values $\tilde{r}_{e,\ell}$ for the sharing amongst the n parties, which are placed in the proof transcript. This means the proof sizes (indeed the proof) for the HyperCube variant is identical to the original variant. The HyperCube variant does however have a potentially shorter run time.

KKW Protocol Phase 2 HyperCube Variant

Phase 2: Second Commitment

1. $J \leftarrow \mathcal{H}_5(h)$.
2. For $e \in J$:
   (a) $\mathbf{y}_e \leftarrow \mathbf{b} \oplus \mathbf{r}_e$.
   (b) For $\ell \in [1, \ldots, B]$:
      i. $[b_{e,\ell}] \leftarrow [r'_{e,\ell}] + \mathbf{y}_e^{(\ell)} - 2 \cdot \mathbf{y}_e^{(\ell)} \cdot [r'_{e,\ell}]$. This linear operation on shares, makes $[b_{e,\ell}]$ a sharing of the input bit $\mathbf{b}^{(\ell)}$.
      ii. For $v \in [1, \ldots, D]$ and $k \in [1, \ldots, N]$ compute $[[b_{e,\ell}]]^{(v,k)}$ via equation (46).
   (c) For each $v \in [1, \ldots, D]$ let $\{[[s_{e,j}]]^{(v,\cdot)}\}_{j=1}^d$ denote the sharing of the values in the subset–sum statement being proved. Recall this will be a linear function of the values $\{[[b_{e,\ell}]]^{(v,\cdot)}\}_{\ell=1}^B$. i.e. we set, for $j = 1, \ldots, d$,

$$[[s_{e,j}]]^{(v,\cdot)} \leftarrow \sum_{\ell=1}^B \mathbf{A}_{j,\ell} \cdot [[b_{e,\ell}]]^{(v,\cdot)}.$$

   (d) $h'_e \leftarrow \mathcal{H}_3\left(\mathbf{y}_e \;\|\; \{\{\{[[s_{e,j}]]^{(v,i)}\}_{j=1}^d\}_{v=1}^D\}_{i=1}^N\right)$.
3. $h' \leftarrow \mathcal{H}_4(\{h'_e\}_{e \in J})$.
4. $D \leftarrow TreePRG.Punc(seed, \lceil \log_2 M \rceil, J)$:
5. $prf_1 \leftarrow prf_1 \;\|\; h' \;\|\; D$.

Figure 101: KKW Protocol Phase 2 HyperCube Variant.

KKW Protocol HyperCube Variant – Verification

$MPCitHead\text{-}Verify\text{-}2((A, \mathbf{s}), prf)$:

Steps 1–6 are the same as in Figure 100.

7. For $e \in J$ and e accepted:
   (a) $\{seed_{e,i}\}_{i \neq l_e} \leftarrow TreePRG.GenPunc(D_e, \{l_e\}, \lceil \log_2 n \rceil)$.
   (b) For $i \neq l_e$:
      i. $comm_{e,i} \leftarrow Commit(seed_{e,i}; \rho_{e,i})$.
      ii. Compute $[r_{e,\ell}]^{(i)}$ using $seed_{e,i}$ just as in Step 3(d)iv of Figure 98.
   (c) For $\ell \in [1, \ldots, B]$:
      i. $\Delta_{e,\ell} \leftarrow \sum_{i \neq l_e} [r_{e,\ell}]^{(i)} - \tilde{r}_{e,\ell}$.
   (d) $h_e \leftarrow \mathcal{H}_1\left(\{\Delta_{e,i}\}_{i=1}^B \;\|\; \{comm_{e,i}\}_{i=1}^n\right)$.
   (e) Let $\mathbf{l} = (l_1, \ldots, l_D)$ denote the D–dimensional vector associated to the value l_e.
   (f) For $i \neq l_e$:
      i. Compute $[r'_{e,\ell}]^{(i)}$, using the $\Delta_{e,i}$ values, just as in Steps 3e of Figure 98.
      ii. Compute $[b_{e,\ell}]^{(i)}$, using $\mathbf{y}_e$, just as in Step 2(b)i of Figure 101.
      iii. Compute $[[b_{e,\ell}]]^{(v,k)}$, for all $\ell \in [1, \ldots, B]$, $v \in [1, \ldots, D]$ and $k \neq l_v$, just as in Step 2(b)ii of Figure 101.
      iv. Compute $[[s_{e,j}]]^{(v,k)}$ for all $j = 1, \ldots, d$, $v \in [1, \ldots, D]$ and $k \neq l_v$, using Step 2c of Figure 101.
   (g) Compute $[[s_{e,j}]]^{(v,l_v)}$ for all $j = 1, \ldots, d$ and $v \in [1, \ldots, D]$ by subtracting the partial sums $\sum_{k \neq l_v} [[s_{e,j}]]^{(v,k)}$ from the actual subset–sum values. In other words compute

$$[[s_{e,j}]]^{(v,l_v)} \leftarrow s_j - \sum_{k \neq l_v} [[s_{e,j}]]^{(v,k)}.$$

   (h) $h'_e \leftarrow \mathcal{H}_3\left(\mathbf{y}_e \;\|\; \{\{\{[[s_{e,j}]]^{(v,i)}\}_{j=1}^d\}_{v=1}^D\}_{i=1}^N\right)$.

Steps 8 and 9 are identical to those in Figure 100.

Figure 102: KKW Protocol HyperCube Variant – Verification.

7.6.4.3 Method 3: Shamir Based Proofs: This variant only is specifically targeted a values of q for which one does not need a Galois Ring extension in order to define Shamir Secret Sharing. In the context of this document this is when $q = p_1 \cdot p_t$ for prime values p_i with $p_i \geq n$. The proof technique below was presented in [FR23] in the context of prime q; that it works for primes of the above form is immediate.

In this section we let $\langle x \rangle$ denote a degree t Shamir sharing of the value $x \in \mathbb{Z}/(q)$ amongst n players. We utilize n and t here, as opposed to our $\mathfrak{n}$ and $\mathfrak{t}$ of our main MPC protocols, as these n players (with a threshold of t) are not actual players in a protocol, but are hyper–parameters of the proof system. We assume in what follows that n is smaller than the smallest prime factor of q, and hence we can take $\{0, 1, \ldots, n\}$ as an exceptional set in $\mathbb{Z}/(q)$.

The share values $(\langle x \rangle_1, \ldots, \langle x \rangle_n)$ will be derived in our protocol from a XOF, via the call

$$(\langle x \rangle_1, \ldots, \langle x \rangle_n) \leftarrow \textit{XOF-Share}(x, \textit{XOF})$$

to the function *XOF-Share*$(x, \textit{XOF})$ in Figure 103.

XOF-Share$(x, \textit{XOF})$

XOF-Share$(x, \textit{XOF})$
    1. $\mathbf{f} \leftarrow \textit{XOF.Next}(t + 1, q)$.
    2. For $i \in [1, \ldots, n]$ compute $\langle x \rangle_i \leftarrow \sum_{j=1}^{t+1} f_j \cdot i^{j-1}$.
    3. Return $(\langle x \rangle_1, \ldots, \langle x \rangle_n)$

Figure 103: Subroutine *XOF-Share*$(x, \textit{XOF})$..

We let $\mathcal{H}_1, \ldots, \mathcal{H}_5, \mathcal{H}_7$ denote six independent collision resistant hash functions. The functions $\mathcal{H}_1, \ldots, \mathcal{H}_5$ are as before, whilst $\mathcal{H}_7$ is defined by

· The function $\mathcal{H}_7$ will take an input string and outputs τ random subsets of $\{1, \ldots, n\}$ of size $t + 1$. The pseudo–code for the function $\mathcal{H}_7$ is given in Figure 104 and protocol for the proof in Figure 105 and Figure 106.

Hash Function $\mathcal{H}_7$

$\mathcal{H}_7(x)$: Hash x into τ random elements of $\{1, \ldots, n\}^{t+1}$
    1. $y \leftarrow \textit{Hash}^{2 \cdot sec}(\textit{DSep}(KKW7)\|x)$.
    2. $\textit{XOF.Init}(y, \textit{DSep}(KKW71))$.
    3. For $i \in [1, \ldots, \tau]$ do:
        (a) $S_i \leftarrow \textit{RandElem}(\textit{XOF}, n, t + 1)$.
    4. Return $(S_1, \ldots, S_\tau)$.

Figure 104: $\mathcal{H}_7$.

In line 2c of Figure 105 the prover only needs to compute the share values for players $1, \ldots, t$. Then the $(t + 1)$'th players share can be deduced from the statement itself, via the reconstruction equation for player $1, \ldots, t + 1$. Finally the shares for players $t + 2, \ldots, n$ can be deduced from the uniqueness of the Shamir sharing given $t + 1$ players shares. Thus this step only requires (essentially) t effort and not n on the part of the prover.

Shamir Based KKW Protocol for Subset Sums – Proof Generation

MPCitHead-Prove-3$((A, \mathbf{s}), \mathbf{b})$:
    Where $\mathbf{s} \in (\mathbb{Z}/(q))^d$ and $\mathbf{A} \in (\mathbb{Z}/(q))^{d \times B}$ are public, and $\mathbf{b} \in \{0, 1\}^B$ is private.

Phase 1: First Commitment

1. $seed \leftarrow \{0, 1\}^{sec}$.
2. $\{seed_i\}_{i=1}^{M} \leftarrow TreePRG.Gen(seed, \lceil \log_2 M \rceil)$.
3. For $e \in [1, \ldots, M]$:
   - (a) *XOF.Init*$(seed_e, DSep(KKW3))$.
   - (b) $\mathbf{r}_e \leftarrow XOF.Next(B)$. These are the input masks for the bits, we think of $\mathbf{r}_e \in \{0, 1\}^B$.
   - (c) $(\langle \mathbf{r}_e \rangle_1, \ldots, \langle \mathbf{r}_e \rangle_n) \leftarrow XOF\text{-}Share(r_e, XOF)$.
   - (d) For $i \in [1, \ldots, n]$:
     - i. $\rho_{e,i} \leftarrow XOF.Next(sec)$.
     - ii. $comm_{e,i} \leftarrow Commit(\langle \mathbf{r}_e \rangle_i; \rho_{e,i})$.
   - (e) $h_e \leftarrow \mathcal{H}_1(\{comm_{e,i}\}_{i=1}^{n})$.
4. $h \leftarrow \mathcal{H}_2(h_1, \ldots, h_M)$.
5. $prf_1 \leftarrow h$.

Phase 2: Second Commitment

1. $J \leftarrow \mathcal{H}_5(h)$. These are the executions we open in the C&C
2. For $e \in J$:
   - (a) $\mathbf{y}_e \leftarrow \mathbf{b} \oplus \mathbf{r}_e$.
   - (b) For $\ell \in [1, \ldots, B]$:
     - i. $\langle \mathbf{b}_e^{(\ell)} \rangle \leftarrow \langle \mathbf{r}_e^{(\ell)} \rangle + \mathbf{y}_e^{(\ell)} - 2 \cdot \mathbf{y}_e^{(\ell)} \cdot \langle \mathbf{r}_e^{(\ell)} \rangle$. This linear operation on shares, makes $\langle \mathbf{b}_e^{(\ell)} \rangle$ a sharing of the input bit $\mathbf{b}^{(\ell)}$.
   - (c) Let $\{\langle s_{e,j} \rangle\}_{j=1}^{d}$ denote the sharing of the values in the subset–sum statement being proved. Recall this will be a linear function of the share values $\{\langle \mathbf{b}_e^{(\ell)} \rangle\}_{\ell=1}^{B}$.

$$\langle s_{e,j} \rangle \leftarrow \sum_{\ell=1}^{B} \mathbf{A}_{j,\ell} \cdot \langle \mathbf{b}_e^{(\ell)} \rangle.$$

   - (d) $h_e' \leftarrow \mathcal{H}_3\left(\mathbf{y}_e \,\|\, \{\{\langle s_{e,j} \rangle_i\}_{j=1}^{d}\}_{i=1}^{n}\right)$.
3. $h' \leftarrow \mathcal{H}_4(\{h_e'\}_{e \in J})$.
4. $D \leftarrow TreePRG.Punc(seed, \lceil \log_2 M \rceil, J)$:
5. $prf_1 \leftarrow prf_1 \,\|\, h' \,\|\, D$.

Phase 3: Completion of Proof

1. $L \leftarrow \mathcal{H}_7(prf_1)$. Write $L = \{(l_e^1, \ldots, l_e^{t+1})\}_{e \in J}$.
2. For $e \in J$:
   - (a) $aux_e \leftarrow \mathbf{y}_e$.
   - (b) For $i \in [1, \ldots, t]$:
     - i. $aux_e \leftarrow aux_e \,\|\, \rho_{e,i} \,\|\, \langle \mathbf{r}_e \rangle_{l_e^i}$.
   - (c) For $i \notin [1, \ldots, t]$:
     - i. $aux_e \leftarrow aux_e \,\|\, comm_{e, l_e^i}$.
3. $prf \leftarrow prf \,\|\, \{aux_e\}_{e \in J}$.
4. Return prf.

Figure 105: Shamir Based KKW Protocol for Subset Sums – Proof Generation.

Shamir Based KKW Protocol for Subset Sums – Verification

MPCitHead-Verify-3$((A, \mathbf{s}), prf)$:
1. Parse the proof *prf* into it's constituent parts.
2. $J \leftarrow \mathcal{H}_5(h)$.
3. $L \leftarrow \mathcal{H}_7(prf_1)$. Write $L = \{(l_e^1, \ldots, l_e^{t+1})\}_{e \in J}$.
4. $\{seed_e\}_{e \notin J} \leftarrow TreePRG.GenPunc(D, J, \lceil \log_2 M \rceil)$.
5. For $e \notin J$ compute h_e as in Step 3 of Phase 1 of Figure 105 using $seed_e$.
6. For $e \in J$:
    (a) For $i \in [1, \ldots, t]$:
        i. Recover ρ_{e,l_e^i} and $\langle \mathbf{r}_e \rangle_{l_e^i}$ from the proof.
        ii. $comm_{e,l_e^i} \leftarrow Commit(\langle \mathbf{r}_e \rangle_{l_e^i}; \rho_{e,l_e^i})$.
    (b) For $i \notin [1, \ldots, t]$:
    (c) Recover $comm_{e,l_e^i}$ from the proof.
    (d) $h_e \leftarrow \mathcal{H}_1(\{comm_{e,i}\}_{i=1}^n)$.
    (e) For $i \in [1, \ldots, t]$:
        i. Compute $\langle \mathbf{b}_e^{(\ell)} \rangle_{l_e^i}$, using $\mathbf{y}_e$, just as in Step 2b of Phase 2 of Figure 105.
        ii. Compute $\langle s_{e,j} \rangle_{l_e^i}$ for $j = 1, \ldots, d$ using the method as in Step 2c of Phase 2 of Figure 105.
    (f) Compute $\langle s_{e,j} \rangle_{l_e^{t+1}}$ from the actual subset–sum values and the shares $\langle s_{e,j} \rangle_{l_e^i}$ for $i = 1, \ldots, t$.
    (g) Compute $\langle s_{e,j} \rangle_{l_e^i}$, for $i > t + 1$, from the shares $\langle s_{e,j} \rangle_{l_e^i}$ for $i = 1, \ldots, t + 1$.
    (h) $h'_e \leftarrow \mathcal{H}_3 \left(\mathbf{y}_e \parallel \{\{[s_{e,j}]^{(i)}\}_{j=1}^d\}_{i=1}^n \right)$.
7. If $h' \neq \mathcal{H}_4(\{h'_e\}_{e \in J})$ then reject the proof.
8. If $h \neq \mathcal{H}_2(h_1, \ldots, h_M)$ then reject the proof.

Figure 106: Shamir Based KKW Protocol for Subset Sums – Verification.

183

8 | Security analysis

To make it easier to follow this section we adopt exactly the same subsection numbering as we did in Section 7. In addition to covering security aspects, we also cover correctness aspects of the protocols where are they not immediate. If the results for a particular section are well known, or in the literature, we simply refer to the literature for the resulting justification.

8.1 Layer Zero

8.1.1 Galois Rings

As mentioned in Section 4.6 most of the results and definitions of this section can be found in references such as [Feh98] and [ACD$^+$19]. The (relatively trivial) result which is perhaps not stated explicitly elsewhere is that we can take the subset $GR(p, F)$ as an exceptional sequence of maximal length in the Galois Ring $GR(q, F)$.

Lemma 11. *The set $GR(p, F)$ (as a set) is a maximal exceptional sequence in $GR(q, F)$.*

Proof. To see that this set is **an** exceptional sequence, notice that since $\mathbb{F}_{p^{\mathfrak{d}}}$ is a finite field we have that every element is invertible, thus for $\alpha_i, \alpha_j \in GR(p, F) \subset GR(q, F)$

$$\alpha_i - \alpha_j \not\equiv 0 \pmod{p}.$$

When $q = p^k$ this means that the inverse of $\alpha_i - \alpha_j$ exists in $GR(q, F)$ for any k, by "Hensel Lifting" of the inverse modulo p. When $q = p_1 \cdots p_k$ this means the inverse of $\alpha_i - \alpha_j$ exists in $GR(q, F)$, and since p is the smallest prime dividing q, we have that $\alpha_i - \alpha_j$ cannot be divisible by any other prime dividing q.

To see that $GR(p, F)$ (as a set) is a maximal exceptional sequence in $GR(q, F)$ consider any sequence of size greater than $p^{\mathfrak{d}}$ lying in $GR(q, F)$. By definition this will contain two elements α_i and α_j such that $\alpha_i \equiv \alpha_j \pmod{p}$. In which case $\alpha_i - \alpha_j$ is not invertible in $GR(q, F)$. We call $GR(p, F)$ (as a set) the **canonical maximal exceptional sequence** in $GR(q, F)$. From now on, for implementation purposes, we will make the choice that $\mathcal{E} \subseteq GR(p, F) \setminus \{0\}$ and $|\mathcal{E}| = \mathfrak{n}$. $\qquad\square$

8.1.2 Reed–Solomon Codes over Galois Rings

The results of Section 7.1.2 can be found in a number of places. For example [ACD$^+$19] contains most of the material on Reed–Solomon codes over Galois Rings, including the generalization of the Berlekamp–Welch algorithm. The syndrome decoding method for traditional view Reed–Solomon coding, comes from [Hal15][Chapter 5]; which is described for fields, but the proofs in that document are seen to easily carry over to the case of Galois Rings. As remarked in an earlier section one can consider our algorithms in this section as a simplification of the more general presentation given in [GTLBNG21] or [QBC13].

We note the following proof, which was omitted from Section 7.1.2 for readability issues.

Lemma 12. *We have $\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{d}}$ if and only if $S_{\mathbf{c}}(Z) = 0$.*

Proof.

$$
\begin{aligned}
\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{d}} \iff S_{\mathbf{c}}(Z) &\equiv \sum_{i=1}^{\mathfrak{n}} \frac{c_i}{L_i(\alpha_i) \cdot (1 - \alpha_i \cdot Z)} \pmod{Z^r} \\
&\equiv \sum_{i=1}^{\mathfrak{n}} \frac{c_i}{L_i(\alpha_i)} \cdot \frac{1}{(1 - \alpha_i \cdot Z)} \pmod{Z^r} \\
&= \sum_{i=1}^{\mathfrak{n}} \frac{c_i}{L_i(\alpha_i)} \cdot \left(\sum_{j=0}^{r-1} (\alpha_i \cdot Z)^j \right)
\end{aligned}
$$

$$= \sum_{j=0}^{r-1} \left(\sum_{i=1}^{\mathfrak{n}} \frac{c_i \cdot \alpha_i^j}{L_i(\alpha_i)} \right) \cdot Z^j$$
$$= 0.$$

$\square$

8.1.2.1 Error Correction over Finite Fields via Berlekamp–Welch: This is standard. See, for example, [BW86]

8.1.2.2 Error Correction over Finite Fields via Gao: Again, this is also standard. See, for example, [Gao03].

8.1.2.3 Error Correction over Rings via Berlekamp–Welch/Gao: The method given in Figure 31 for $q = p_1 \cdots p_k$ is an immediate application of undergraduate algebra. The lifting procedure in Figure 31 for the case of $q = p^k$ appears as Figure 1 in [ACD$^+$19] and as Algorithm 2 in [QBC13]; we repeat the correctness result here for completeness.

Lemma 13. *When $q = p^k$ the algorithm in Figure 31 corrects a received vector into a codeword assuming the number of errors is bounded by $e \leq \lfloor (\mathfrak{n} - s - \mathfrak{v})/2 \rfloor$, where s is the number of erasures in the received vector.*

Proof. To simplify the exposition assume $s = 0$ for the received input vector. Suppose the underlying codeword is generated from the polynomial $f(Z) \in R[Z]$ of degree less than $\mathfrak{v}$. We can write $f(Z)$ in terms of it's p–adic expansion up to precision k, i.e.

$$f(Z) = f_0(Z) + p \cdot f_1(Z) + p^2 \cdot f_2(Z) + \cdots + p^{k-1} \cdot f_{k-1}(Z),$$

where $f_i(Z)$ is a polynomial with coefficients in the range $[0, \ldots, p-1]$ of degree less than $\mathfrak{v}$. The received vector should be given by

$$c_i \equiv f(\alpha_i) \pmod{q},$$

however, it is wrong in e positions, where $e \leq \lfloor (\mathfrak{n} - \mathfrak{v})/2 \rfloor$. Obviously we can find $f_0(Z)$ by applying the standard Berlekamp–Welch/Gao algorithm over the finite field. Having found $f_0(Z)$ we produce the vector

$$\mathbf{y} = \mathbf{c} - \mathbf{t}$$

where $t_i \equiv f_0(\alpha_i) \pmod{q}$. Note that, we evaluate $f_0(\alpha_i)$ over the full ring and not modulo p here. The vector $\mathbf{y}$ will be divisible by p for all positions where there was no error modulo p, if there is a position which is not divisible by p then this corresponds to an error position, and we will treat it as an erasure position in the next loop. Thus by dividing $\mathbf{y}$ by p, we have, for all positions for where there is no error,

$$y_i/p = (c_i - t_i)/p = (f(\alpha_i) - f_0(\alpha_i))/p$$
$$\equiv f_1(\alpha_i) + p \cdot f_2(\alpha_i) + \cdots + p^{k-2} \cdot f_{k-1}(\alpha_i) \pmod{p^{k-1}}.$$

Thus, we can recover $f_1(Z)$ by applying the standard Berlekamp–Welch algorithm over finite fields to the vector $\mathbf{z} \leftarrow (\mathbf{y}/p) \pmod{p}$. By repeating this k times we recover the corrected vector. $\square$

8.1.2.4 Syndrome Decoding: As noted before the method in Section 7.1.2.4 is derived from the presentation in [Hal15][Chapter 5]. We start with explaining syndrome decoding over the finite field $GR(p, F) = \mathbb{F}_{p^{\mathfrak{v}}}$, see Figure 32. Our method, to produce the correction value, takes as input the syndrome

polynomial, $S_{\mathbf{c}}(Z)$; plays a crucial role in this method. Suppose we have that $\mathbf{c} \in RS_{\mathfrak{n},\mathfrak{d}}$, then we know that $S_{\mathbf{c}}(Z) = 0$. However if $\mathbf{e}$ is an error vector then we have

$$S_{\mathbf{r}}(Z) = S_{\mathbf{c}+\mathbf{e}}(Z) = S_{\mathbf{c}}(Z) + S_{\mathbf{e}}(Z) = S_{\mathbf{e}}(Z),$$

for the received vector $\mathbf{r} = \mathbf{c} + \mathbf{e}$. Now if we define B to be the set of error locations, i.e. $B = \{i : e_i \neq 0\}$ then we have

$$S_{\mathbf{e}}(Z) \equiv \sum_{b \in B} \frac{e_b}{L_b(\alpha_b) \cdot (1 - \alpha_b \cdot Z)} \quad (\text{mod } Z^r).$$

Thus $S_{\mathbf{e}}(Z)$ encodes the error locations B, as well as the error values e_b. Our goal is then to extract these values from the syndrome polynomial. The syndrome polynomial encodes both the error positions, and the error values via the equation

$$\sigma(Z) \cdot S_{\mathbf{e}}(Z) \equiv \omega(Z) \quad (\text{mod } Z^r) \tag{47}$$

where

$$\sigma(Z) = \prod_{b \in B}(1 - \alpha_b \cdot Z), \tag{48}$$

$$\omega(Z) = \sum_{b \in B} \frac{e_b}{L_b(\alpha_b)} \cdot \left(\prod_{a \in B \setminus \{b\}} (1 - \alpha_a \cdot Z) \right). \tag{49}$$

If equation (47) has a unique solution then we can determine the unique error vector which resulted in us receiving $\mathbf{c} + \mathbf{e}$ instead of $\mathbf{c}$.

Following the arguments in Chapter 5 of [Hal15] we see that there is at most one solution $(\sigma(Z), \omega(Z))$ of this equation with the following conditions satisfied:

$$deg(\sigma(Z)) \leq r/2,$$
$$deg(\omega(Z)) < r/2,$$
$$\sigma(0) = 1,$$
$$\gcd(\sigma(Z), \omega(Z)) = 1.$$

Once one has determined $\sigma(Z)$ and $\omega(Z)$ then one can determine the error locations and values by working out which values of α_i are such that $\sigma(1/\alpha_i) = 0$ etc.

One could solve equation (47) much like we solved the equation in Figure 29 above; i.e. by creating a linear system with r equations in the r unknown coefficients of $\sigma(Z)$ and $\omega(Z)$. We can solve this linear system modulo p, and then, given the polynomials $\sigma(Z)$ and $\omega(Z)$, we can compute the error vector $\mathbf{e}$, and correct the received vector into a valid codeword. A more elegant solution, for finite fields, is to use the extended Euclidean Algorithm, as we have in Figure 32, again see [Hal15][Chapter 5] for the full justification of this method.

The method in Figure 33 for the case of syndrome decoding over Galois Rings when $q = p^k$ can be justified as follows: Consider as input the received vector $\mathbf{r} = \mathbf{c} + \mathbf{e}$. By definition we know that we have, in $GR(q, F)$, that $\mathbf{c} = M_{\mathfrak{n},\mathfrak{d}} \cdot \mathbf{f}$, for some vector $\mathbf{f}$ of dimension $\mathfrak{d}$. In the first iteration of the algorithm $SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$, the vector $\mathbf{e}^{(0)}$ clearly corresponds to the value of $\mathbf{e}$ (mod p). The algorithm then proceeds to compute

$$S_{\mathbf{e}}^{(1)}(Z) \leftarrow S_{\mathbf{e}}^{(0)}(Z) - S_{\mathbf{e}^{(0)}}(Z) = S_{\mathbf{e}}(Z) - S_{\mathbf{e}^{(0)}}(Z).$$

This is the syndrome polynomial which would have been computed if we had started with the received vector $\mathbf{r}^{(1)} \leftarrow \mathbf{c} + \mathbf{e} - \mathbf{e}^{(0)}$. We know that $\mathbf{e} \equiv \mathbf{e}^{(0)}$ (mod p), thus we know that $S_{\mathbf{e}^{(1)}}(Z)$ is divisible by p

(so the division by p in line 1(c)i of the next iteration of the loop will be valid). We have also (implicitly) determined $\mathbf{c}^{(0)} = \mathbf{c} \pmod{p}$, and we know $\mathbf{c}^{(0)} = M_{\mathfrak{n},\mathfrak{v}} \cdot \mathbf{f}^{(0)}$ for some vector $\mathbf{f}^{(0)} = \mathbf{f} \pmod{p}$, i.e. $\mathbf{f} = \mathbf{f}^{(0)} + p \cdot \mathbf{g}$ for some vector $\mathbf{g} \in RS(p^{k-1}, F)$. Hence,

$$\mathbf{c} - \mathbf{c}^{(0)} = M_{\mathfrak{n},\mathfrak{v}} \cdot (\mathbf{f} - \mathbf{f}^{(0)})$$
$$= p \cdot M_{\mathfrak{n},\mathfrak{v}} \cdot \mathbf{g}$$

Thus our algorithm is working (implicitly), at the next iteration, with the vector

$$(\mathbf{c} - \mathbf{c}^{(0)})/p = M_{\mathfrak{n},\mathfrak{v}} \cdot \mathbf{g},$$

the algorithm will be correct since this vector is also a codeword for the code. Thus we can recurse with our algorithm to obtain the desired solution modulo q.

The method for the case of $q = p_1 \cdots p_k$ is an immediate application of the Chinese Remainder Theorem.

8.1.3 Shamir Secret Sharing over Galois Rings

The methods described in Section 7.1.3 are all standard, see for example [Feh98] and [ACD$^+$19].

8.1.3.1 Error Detection: Again this is standard given the above results on Reed–Solomon codes.

8.1.3.2 Error Correction: Again this is standard given the above results on Reed–Solomon codes.

8.1.3.3 Randomness Extraction: The method in Section 7.1.3.3 is a standard result for secret sharing over finite fields, which was generalized to Galois Rings in [ACD$^+$19].

8.1.4 Commitment Schemes

Our commitment scheme given in Figure 35 appears to be a folklore construction; we could not find a good a reference for it in the literature. To break the binding property of the commitment scheme the adversary needs to come up with a collision in the hash function $Hash^{2 \cdot sec}(DSep(COMM)\|m\|r) = Hash^{2 \cdot sec}(DSep(COMM)\|m'\|r')$. To break hiding the adversary needs to invert the hash function. However, there could be many pre–images for a specific image, thus even inverting the hash function may not result in determining the precise committed pre–image. Both collision resistance and pre–image resistance are implied by the fact that the hash function is modelled as a random oracle, and the fact that the output size is of bit–length $2 \cdot sec$.

8.1.5 Bit Generation

The bit generation methods of Section 7.1.5 are all relatively standard algorithms from the literature.

8.1.5.1 *Solve*(v): This algorithm is an application of standard results in finite fields (see for example [BSS99], page 26]) combined with a standard application of Hensel Lifting.

8.1.5.2 *Sqrt*(v, p): As remarked these are classical algorithms which can be found in [Coh93, Sho05].

8.1.5.3 Random Bits p Odd Prime: As remarked in the text this method seems to have been first proposed in [DKL$^+$13].

8.1.5.4 Random Bits q Power of Two: Again as remarked in the text this seems to have been first proposed in [OSV20], with a simplification presented in [DDE$^+$23].

8.1.6 TreePRG

As explained in the text in Section 7.1.6, the algorithm in Figure 40 is adapted from the presentation in the full version of [GGHAK22] and that given in [NNL01].

8.2 Layer One

8.2.1 Broadcast

As explained in Section 7.2.1 our protocol in Figure 42 is based on modifications to Bracha's proto–col [Bra87], which are taken from https://hackmd.io/@alxiong/bracha-broadcast. Intuitively the protocol is secure for the following reasoning (which can be made formal if desired following the ar–guments in the references above): We have $\mathfrak{n}$ parties, of which $\mathfrak{t}$ at most are malicious, with $\mathfrak{n} > 3 \cdot \mathfrak{t}$. Consider a message m such that no honest party has echoed it. In this case, in round two, an honest party sees at most $\mathfrak{t}$ echos on m. Thus in round three no honest party will vote for m as an honest party will only vote if it sees $\mathfrak{n} - \mathfrak{t}$ echos and $\mathfrak{t} < \mathfrak{n} - \mathfrak{t}$ for $\mathfrak{n} < 3 \cdot \mathfrak{t}$. Thus no honest party will ever vote for m, because if no honest party has ever voted for m in a previous round, an honest party never sees more than $\mathfrak{t}$ votes on m, and $\mathfrak{t} < \mathfrak{t} + r - 3$ for $r \geq 4$. Finally, if no honest party has ever voted for m, no honest party will output m. So for honest parties to agree on a message m, some honest party must have echoed it, and by definition of "honest", all the (honest) parties will have seen this echo.

8.2.2 Dispute Control

The dispute control framework description is taken from [DN07]. The modification of the broadcast protocol *Synch-Broadcast*(S, m) to obtain *Synch-Broadcast*($S, m, Corrupt$) of Figure 43 is immediate.

8.2.3 Robust Opening

The algorithm *RobustOpen* in Figure 44 is a classical method. The method for asynchronous networks and $d = \mathfrak{t} < \mathfrak{n}/3$ is called "online error correction", and was first presented in [BCG93]. Assuming the input sharing is of an element in $\mathbb{Z}/(q)$ then the robust open protocol in Figure 44 will output the value in $\mathbb{Z}/(q)$, even if adversarial parties introduce errors. This is despite the shares themselves, and the Lagrange interpolation coefficients, being defined by elements in the Galois Ring.

8.2.3.1 Optimization via the King Paradigm: The protocol in Figure 45 is a standard method, which first appeared for the case of Shamir sharing over finite fields in [DN07]. That this optimization from [DN07] also applies to the Galois Ring case follows from [ACD$^+$19].
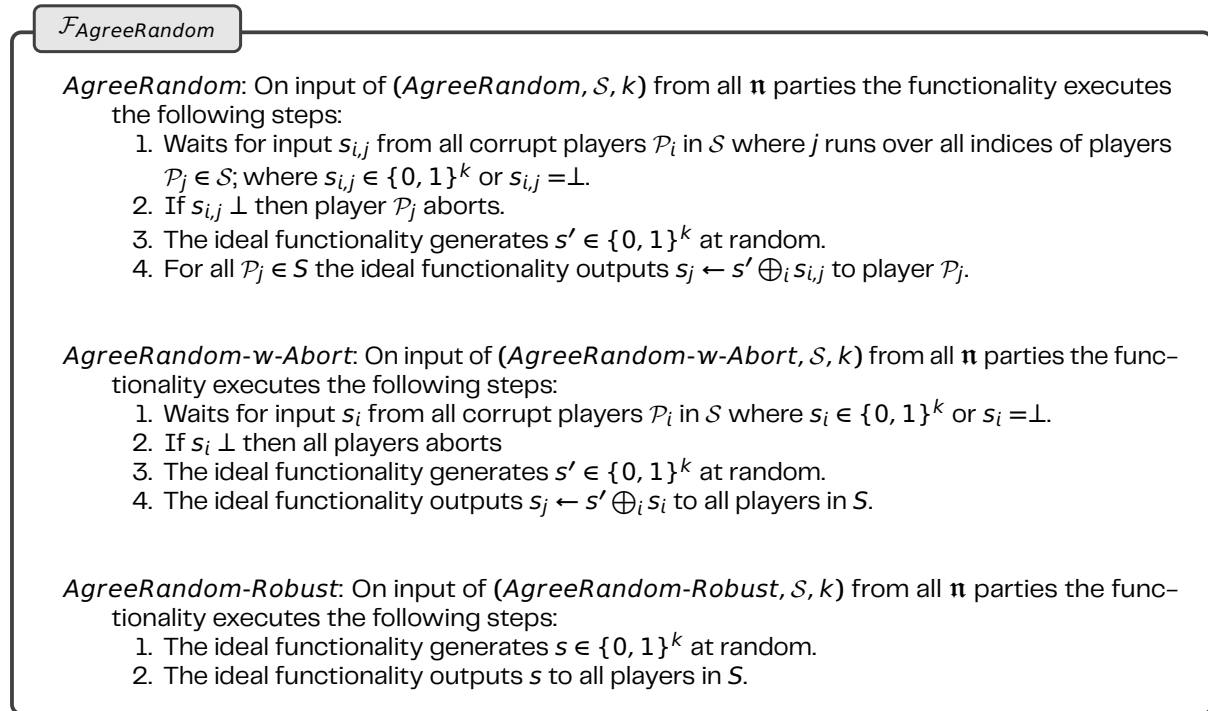
8.2.4 Verifiable Secret Sharing

As marked in Section 7.2.4, the protocol *VSS*($\mathcal{P}_k, s, \mathfrak{t}, Corrupt$) in Figure 46 first appeared in the case of sharing over finite fields in [GIKR01]. That the protocol, and it's proof (given in [GIKR01]), generalizes from finite fields to Galois Rings is immediate.

8.2.5 Agree on a Random Number

The methods *AgreeRandom*($\mathcal{S}, k$), *AgreeRandom-w-Abort*($\mathcal{S}, k$) and *AgreeRandom-Robust*($\mathcal{S}, k, \langle r \rangle$) of Section 7.2.5 are all relatively standard protocols which enable subsets of parties to agree on common

random values amongst themselves. That the protocols implement the ideal functionalities given in Figure 107 is easy to see.

$\mathcal{F}_{AgreeRandom}$

AgreeRandom: On input of (*AgreeRandom*, $\mathcal{S}, k$) from all $\mathfrak{n}$ parties the functionality executes the following steps:
 1. Waits for input $s_{i,j}$ from all corrupt players $\mathcal{P}_i$ in $\mathcal{S}$ where j runs over all indices of players $\mathcal{P}_j \in \mathcal{S}$; where $s_{i,j} \in \{0,1\}^k$ or $s_{i,j} =\perp$.
 2. If $s_{i,j} \perp$ then player $\mathcal{P}_j$ aborts.
 3. The ideal functionality generates $s' \in \{0,1\}^k$ at random.
 4. For all $\mathcal{P}_j \in S$ the ideal functionality outputs $s_j \leftarrow s' \bigoplus_i s_{i,j}$ to player $\mathcal{P}_j$.

AgreeRandom-w-Abort: On input of (*AgreeRandom-w-Abort*, $\mathcal{S}, k$) from all $\mathfrak{n}$ parties the func–tionality executes the following steps:
 1. Waits for input s_i from all corrupt players $\mathcal{P}_i$ in $\mathcal{S}$ where $s_i \in \{0,1\}^k$ or $s_i =\perp$.
 2. If $s_i \perp$ then all players aborts
 3. The ideal functionality generates $s' \in \{0,1\}^k$ at random.
 4. The ideal functionality outputs $s_j \leftarrow s' \bigoplus_i s_i$ to all players in S.

AgreeRandom-Robust: On input of (*AgreeRandom-Robust*, $\mathcal{S}, k$) from all $\mathfrak{n}$ parties the func–tionality executes the following steps:
 1. The ideal functionality generates $s \in \{0,1\}^k$ at random.
 2. The ideal functionality outputs s to all players in S.

Figure 107: The ideal functionality $\mathcal{F}_{AgreeRandom}$.

Strictly speaking to implement the robust version of the functionality we need to combine the protocol in *AgreeRandom-Robust* with a method to generate valid secret sharings of random values. In our protocols this is exactly what we do; we use a VSS for each player to enter a random sharing, then we apply randomness extraction to extract a subset of unknown random sharings, and then we utilize *AgreeRandom-Robust* to open this random value to the required parties.

8.2.6 Generating Random Shamir Secret Sharings

The PRSS methodologies in Section 7.2.6 are all very much standard applications of the original method–ology as introduced in [CDI05]; except for

- Our checking procedures *PRSS.Check* and *PRZS.Check* which we will discuss below.
- The method *PRSS-Mask.Next* which produces a shared value which is a "small" value in $\mathbb{Z}/(q)$. This latter method is derived from ideas given in [CLO+13] and [DDE+23].

8.2.6.1 PRSS Initialization: To perform the initializations of our PRSS keys we simply need to generate a secret key value for every relevant subset. This is done by applying the various methods in Section 7.2.5 to the relevant subsets.

8.2.6.2 PRSS: This is the method described in [CDI05] but with an explicit description of how the relevant *PRF* is constructed via using AES in counter mode. When q is not a power of two, the use of a larger value of v in order to generate a secret shared value, and secret shares, which are statistically close to uniform is standard. That the method *PRSS.Check* is valid follows from the following Lemma.

Lemma 14. *The procedure PRSS.Check recovers the PRSS shares correctly irrespective of any adver–sarial behavior as long as $\mathfrak{t} < \mathfrak{n}/3$.*

Proof. Notice that when $\mathfrak{t} < \mathfrak{n}/3$ the sets A have size $\mathfrak{n} - \mathfrak{t} > 2 \cdot \mathfrak{t}$, thus the set A always contains an honest majority, and hence the method of selecting the correct value in Figure 51 will always produce the correct sharing. This means that *PRSS.Check*($cnt, Corrupt$) will output the correct set of shares, for the counter cnt, irrespective of adversarial behavior.

Also note that, as the sets Ψ_i are broadcast, all players agree on what was sent by all other players. Hence, corrupt players can be identified and agreed upon. This means that in line 4 of *PRSS.Check*, all honest players will agree on $\mathcal{P}_j$ due to the use of the previous reliable broadcast in line 2. The identified corrupt player can then be removed from any further computations. □

8.2.6.3 PRZS: Again this is standard, and the correctness of *PRZS.Check* follows from the analogue of Lemma 14.

8.2.6.4 PRSS–Mask: That the *PRSS-Mask.Next* operation creates the correct output is immediate.

8.2.7 CoinFlip

The *CoinFlip* protocol is standard, the method we present is given in [DNO7]. It is only used within the batched VSS protocols in the *nLarge* setting, which are themselves only used in the offline phase of the MPC protocol in the *nLarge* setting. Thus the *CoinFlip* does not actually implement (directly) any of the commands of the ideal functionalities used to model our MPC protocol, it is only used internally in the implementing protocols.

8.3 Layer Two

The Layer Two of our protocol stack implements the ideal functionality $\mathcal{F}_{Prep}$ given in Figure 108. The set of adversaries in our protocol will be denoted $\mathcal{A}$, we have $\mathcal{A} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathfrak{n}}\}$ and $|\mathcal{A}| \leq \mathfrak{t}$, of which at most $\mathfrak{t}$ are assumed to be statically corrupted by the adversary before the protocol starts. Recall, when reading $\mathcal{F}_{Prep}$ that the routine $MPC^O.Init$ comes in two variants: Either an active–with–abort secure variant or a robustly secure variant. These two variants are captured by the functionality given in Figure 108.

Note in our functionality $\mathcal{F}_{Prep}$ the adversary does not get to select their share values for either the output of *NextRandom* or the output of *GenTriples*. This is because in the protocol the adversaries share values get given to them by either the evaluation of the PRSS (in the case of threshold profile *nSmall*) or the randomness extraction methodology (in the case of threshold profile *nLarge*).

That our protocols emulate this ideal functionality in both the *nSmall* and *nLarge* threshold profiles follows from the security proofs/arguments in the MPC literature; and the results we outline in the following sub–sections. In particular our protocols follow the blueprint of [DNO7]; and the arguments there carry over to our protocols[19] We note that the extension of the proof techniques alluded to in [DNO7] from arbitrary finite field to Galois Rings is immediate, given the previously mentioned results on about the methods and protocols on our Layer Zero and Layer One;

8.3.1 Offline MPC Protocol for Threshold Profile Category *nSmall*

In the case of threshold profile *nSmall* we utilize *PRSS* and *PRZS* evaluations to define much of the functionality $\mathcal{F}_{Prep}$. We have already remarked that these are all secure given standard arguments dating back to [CDI05]. The offline protocol for *nSmall* uses our novel checking procedure, which we show to be correct and secure in the following theorem.

Theorem 4. *The offline protocol in Figure 57 is both private and robust.*

---

[19]The paper [DNO7] does not provide an explicit security proof of their protocol. They claim UC security of their protocol, but state "We will however not prove this with full simulation proofs in the following, as the security of our protocols follow using standard proof techniques." We follow the same approach here in saying that it all follows "using standard proof techniques."

Init(): On receiving (*Init*) from all parties:
1. If the robustly secure variant then this functionality is a No-Op.
2. If the active–with–abort secure variant is desired, then the functionality waits for input from the adversary. On receiving this input, if the adversary says *abort* then the functionality aborts, otherwise the functionality returns.

NextRandom: On receiving (*NextRandom, varid$_a$*) from all parties:
1. The ideal functionality generates a uniformly random $a \leftarrow GR(q, F)$.
2. The ideal functionality generates a secret sharing $\langle a \rangle$ of a (for example using algorithm *Share*(a) from Figure 34).
3. The values $\langle a \rangle_i$ for all $\mathcal{P}_i \in \mathcal{A}$ are given to the adversary.
4. The functionality stores the value a into the location with handle $varid_a$.

GenTriples(): On receiving (*GenTriples, varid$_a$, varid$_b$, varid$_c$*) from all parties:
1. The ideal functionality generates a uniformly random $a, b \leftarrow GR(q, F)$.
2. The ideal functionality computes $c = a \cdot b$.
3. The ideal functionality generates secret sharings $\langle a \rangle$, $\langle b \rangle$ and $\langle c \rangle$ (for example by using algorithm *Share*($\cdot$) from Figure 34).
4. The values $(\langle a \rangle_i, \langle b \rangle_i, \langle c \rangle_i)$ for all $\mathcal{P}_i \in \mathcal{A}$ are given to the adversary.
5. The functionality stores the values a, b and c into the locations with handles $varid_a$, $varid_b$ and $varid_c$.

Figure 108: The MPC Offline Ideal Functionality $\mathcal{F}_{Prep}$.

Proof. The protocol is obviously private, what is less obvious is that it is robust.

The main issue we need to deal with in the triple generation protocol is that the error correction on line 1j of Figure 57, may fail when we have more than $\mathfrak{n}/4$ adversaries; since we are trying to error correct a sharing of degree $2 \cdot \mathfrak{t}$. When $\mathfrak{t} < \mathfrak{n}/4$, this will always be correct and we will never leave the happy path. However, when $\mathfrak{t} \geq \mathfrak{n}/4$, the adversaries could force us onto the unhappy processing in line 1l of $MPC^S.GenTriples$ in Figure 57.

To force the players into the unhappy path the adversary needs to introduce $(\mathfrak{t} - s)/2$ errors into the values which have been broadcast. The maximum number of errors which can be introduced by the adversary in a given round is $\mathfrak{t} - s$, since s counts the number of $\bot$ values, and hence the number of players in *Corrupt*, this ability to introduce errors decreases the more errors that the adversary introduces over time as long as our unhappy processing detects which values have been introduced incorrectly.

The unhappy processing of $MPC^S.GenTriples$ in Figure 57 determines the players to add to the set *Corrupt* as follows: Because the opening is broadcast, as opposed to done via point–to–point channels, we know that all parties agree on the values $\{\langle d \rangle_j^{2 \cdot \mathfrak{t}}\}$ opened. So when we perform the check against the values $\langle d' \rangle_j^{2 \cdot \mathfrak{t}}$, which are by definition the same for all parties, then we know (and agree) on who is cheating. Upon agreeing who is cheating we can ignore them on the next iteration.

In line 1j we then have a degree $2 \cdot \mathfrak{t}$ sharing to recover, from n shares of which we know at most $\mathfrak{t}$ are bad, but for which we also know the location of u errors. Thus (in effect) we are executing error correction for a Reed–Solomon code with parameters $\mathfrak{n}' = \mathfrak{n} - u$, $\mathfrak{v} = 2 \cdot \mathfrak{t} + 1$ and with at most $\mathfrak{t} - u$ bad players. Alternatively, we are applying normal Reed–Solomon error correction with we have u erasures and at most $\mathfrak{t} - u$ errors.

The bad parties can make the honest parties pass to the unhappy path at most $\mathfrak{t}$ times, before honest parties would obtain $s = \mathfrak{t}$. At this point there are guaranteed to be no errors, and recovery, in line 1j, will succeed as we are recovering a degree $2 \cdot \mathfrak{t}$ polynomial from $\mathfrak{n} - \mathfrak{t}$ honest shares; which is always possible as $\mathfrak{t} < \mathfrak{n}/3$. □

8.3.2   Offline MPC Protocol for Threshold Profile Category *nLarge*

The blueprint for our offline protocols for threshold profile *nLarge* is almost identical to that followed in [DNO7].

8.3.2.1   A "Batched" Statistical VSS: The only (slightly non–trivial) deviation from [DNO7] is in the method used to check correctness within the procedures *LocalSingleShare* and *LocalDoubleShare* used in threshold profile *nLarge* given in Figure 60 and Figure 61. In particular the variant in [DNO7], finite fields, extends the underlying base field in order to obtain a suitable rejection probability when the base field is small. It turns out that in the case of Galois Rings $GR(p^k, F)$ for a small prime p, parallel repetition is more efficient.

Whilst these two modifications (from finite fields to Galois Rings, and from field/ring extension to parallel repetition) are obvious we feel it appropriate to reprove the statement in relation to security and correctness. The proof follows much the same argument as in [DNO7], and for simplicity, we restrict to a proof for the procedure *LocalSingleShare*, with the proof for *LocalDoubleShare* proceeding in much the same manner, much like in [DNO7].

Theorem 5. *The output of LocalSingleShare are, with probability at least $1 - 2^{-dist}$, valid degree $\mathfrak{t}$ sharings of elements in $s_i \in R$ even in the case of dishonest $\mathcal{P}_i$. If $\mathcal{P}_i$ is honest then no information leaks to dishonest parties.*

Proof. We first prove correctness and then security.

Degree $\mathfrak{t}$: We want to show that the sharings are of degree $\mathfrak{t}$ and not some higher degree.
We let $f_{0,g}(X)$ denote the lowest degree polynomial consistent with the honest players sharings of $\langle r_g \rangle$ and we let $f_j(X)$ denote the lowest degree polynomial consistent with the honest shares of $\langle s_j \rangle$.
We define $F_g(X)$ to be the polynomial

$$F_g(X) = f_{0,g}(X) + \sum_{j=1}^{\ell} x_{j,g} \cdot f_j(X).$$

This polynomial is consistent with the honest shares of $\langle y_g \rangle$, but is not necessarily the lowest degree such polynomial.

If (at least) one of the sharings of s_j or r_g is not of degree $\mathfrak{t}$, then this implies that (at least) one of the polynomials $f_j(X)$ or $f_{0,g}(X)$ is of degree $d > \mathfrak{t}$. Let d be the maximal such degree and write

$$f_{0,g}(X) = \alpha_{0,g} \cdot X^d + \cdots,$$
$$f_j(X) = \alpha_j \cdot X^d + \cdots.$$

The coefficient of $F_g(X)$ of degree d is then given by

$$M_g(Y_1, \ldots, Y_\ell) = \alpha_{0,g} + \sum_{j=1}^{\ell} \alpha_j \cdot Y_j,$$

Thus $M_g(x_{1,g}, \ldots, x_{\ell,g})$ will be zero with probability at most $1/|S|$. Assuming the multinomial M_g is not identically zero for all g, this means there exists an index k (except with probability $1/|S|^m \leq 2^{-dist}$) such that $F_k(X)$ has degree d.

For this value of k we consider $G_k(X)$, the polynomial of lowest degree which is consistent with the honest shares of $\langle y_k \rangle$. Write $H_k(X) = F_k(X) - G_k(X)$. We clearly have that $H_k(\alpha_i) = 0$ for all honest parties $\mathcal{P}_i$ and $\mathcal{H}_{vc}(X)$ has degree at most d, which is less than the number of honest parties by definition. Thus we have that $H_k(X)$ is the zero polynomial.

Hence we have that $G_k(X) = F_k(X)$, and so the honest shares of $\langle y_k \rangle$ are on a polynomial of degree $d > \mathfrak{t}$. This implies a dispute will be generated.

If no dispute is generated this implies (with probability at least $1 - 2^{-dist}$) that all the sharings are of the correct degree $\mathfrak{t}$.

Privacy: If $\mathcal{P}_i$ is honest then the value for each index g the underlying value behind the shares $\sum_{j=1}^{\ell} x_i \cdot \langle s_j \rangle$ is an element of R, which is a function of the secret values $\langle s_j \rangle$. However, this value is then masked information theoretically by adding in $\langle r_g \rangle$, for a uniformly random chosen value of $r_g \in R$. Thus no information leaks to the adversary on reconstruction. □

8.3.2.2 Randomness Extraction: The randomness extraction performed in Figure 62, is exactly an application of what we discussed in Section 8.1.3.3 above.

8.3.2.3 Offline Protocols: The offline protocol in Figure 63 is then identical to that in [DN07] in the case of $\mathfrak{t} < \mathfrak{n}/4$; so we refer to that paper for a discussion of its security and properties.

8.4 Layer Three

8.4.1 Online MPC Protocol for All Threshold Profile Categories

We now present the ideal functionality $\mathcal{F}_{MPC}$ given in Figure 109. The ideal functionality deals with handles on values, it passes the handles (which we denote by $varid_x$ etc) to the players, yet maintains a list of the secret values indexed by these handles in an internal store. The functionality $\mathcal{F}_{MPC}$ acts like the traditional Arithmetic Black Box (ABB) functionality (see for example [KOS16]); except that there is no ability for the players to enter an input value (we do not need that in our protocols). Instead of allowing players to enter their own secret values we only enable players to jointly sample a secret random value.

8.4.1.1 Multiplication: Using standard techniques it is easy to show that the protocols emulating $\mathcal{F}_{MPC}$ (i.e. the multiplication routine om Figure 67) is secure in the $\mathcal{F}_{Prep}$–hybrid model.

8.4.1.2 *MPC.Mask*: In the threshold profile *nSmall* we add an additional procedure into the functionality $\mathcal{F}_{MPC}$ which is given in Figure 110. It is obvious that the implementation in *PRSS-Mask.Next* produces the same distribution of random value as in the functionality command *Mask*. That the leakage of the values (r_A, r'_A) given in the ideal functionality happens also in the implementation follows from the fact that the adversary has the PRSS keys for these values, and hence can determine these summands in the clear.

What is less obvious is that it provides something useful. So we now discuss how it will be used, and the relevant proofs. In a number of places we need to generate a secret shared mask value $\langle E \rangle$, of a value $E \in \mathbb{Z}/(q)$ of a specific size. This mask value will be used to mask another (small) secret shared value $\langle e \rangle$, also with $e \in \mathbb{Z}/(q)$, via addition, i.e. we compute $\langle E + e \rangle$. This masked value $E + e$ is then opened and processed. The idea being that E will be large enough to ensure that $E + e$ contains statistically no information about the value e; whereas $E + e$ is small enough to enable additional useful processing.

So we assume that we have the value $e \pmod{q}$ that we want to mask is of size bounded by Bd, i.e. $|e \pmod{q}| < Bd$, where we take the centred modular reduction. In particular, the value of e is given via a secret sharing $\langle e \rangle$, for which we are guaranteed that $|e \pmod{q}| < Bd$. We wish to construct a secret shared value $\langle E \rangle$ that we will add onto $\langle e \rangle$ where E is bounded in size by

$$|E| < 2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} \cdot Bd = Bd'.$$

Init(): On receiving (*Init*) from all parties:
1. If the robustly secure variant then this functionality is a No–Op.
2. If the active–with–abort secure variant, then the functionality waits for input from the adversary. If the adversary says *abort* then the functionality aborts, otherwise the functionality returns.

NextRandom(): On receiving (*NextRandom*, $varid_a$) from all parties:
1. The ideal functionality generates a uniformly random $a \leftarrow GR(q, F)$.
2. The ideal functionality stores a into the location with handle $varid_a$.

LinearComb(): On receiving (*LinearComb*, $varid_a$, $\{c_i, varid_{x_i}\}_{i=1}^t$, c) from all parties where $c_i, c \in GR(q, F)$:
1. The functionality retrieves the values $\{x_i\}_{i=1}^t$ stored in the locations with handles $\{varid_{x_i}\}_{i=1}^t$.
2. The functionality computes $a \leftarrow c + \sum_{i=1}^t c_i \cdot x_i$.
3. The value a is stored in the location with handle $varid_a$.

Mult(): On receiving (*Mult*, $varid_a$, $varid_b$, $varid_c$) from all parties:
1. The ideal functionality retrieves the values a and b stored in the locations with handles $varid_a$ and $varid_b$.
2. The ideal functionality computes $c \leftarrow a \cdot b$.
3. The value c is stored in the location with handle $varid_c$.

Open(): On receiving (*Open*, $varid_a$, $\mathcal{U}$) from all parties:
1. The functionality obtains the value a stored in the location with handle $varid_a$.
2. The value a is returned to all players in the subset $\mathcal{U}$, with players not in $\mathcal{U}$ obtaining nothing.

Figure 109: The MPC Ideal Functionality $\mathcal{F}_{MPC}$.

Mask: On receiving (*Mask*, $varid_a$, Bd) from all parties:
1. For every set $A \subseteq \{1, \ldots, \mathfrak{n}\}$ of size $\mathfrak{n} - \mathfrak{t}$ generate two random values $r_A, r'_A \leftarrow U(-Bd, Bd)$, i.e. $r_A, r'_A \in (-Bd, \ldots, Bd]$.
2. Set $a \leftarrow \sum_A (r_A + r'_A)$.
3. Assign a to the $varid_a$.
4. Return (r_A, r'_A) to the adversary for every subset A such that there exists an adversarial player $\mathcal{P}_i$ with $i \in A$.

Figure 110: Additional Command for Threshold Profile *nSmall* for Functionality $\mathcal{F}_{MPC}$.

Using *PRSS-Mask.Next* the shared value E is not chosen uniformly in the range $[-Bd', \ldots, Bd']$. Instead it will be chosen as a sum of at most $2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}}$ uniformly random values in the range $[-2^{stat} \cdot Bd, \ldots, 2^{stat} \cdot Bd]$. The adversary has the potential to learn all bar two of these uniform random variables. That this secret shared value $\langle E \rangle$ is not only easy to generate, via a modification to our *PRSS* machinery, it also gives us the following strong statistically security result.

Theorem 6. *To distinguish $e + E$ and E where e is an unknown fixed value of size $|e| < Bd$ and E is chosen as the sum of at least two unknown uniformly chosen random values in the range $[-2^{stat} \cdot Bd, \ldots, 2^{stat} \cdot$*

Bd] *requires* $2^{2 \cdot stat}$ *such samples.*

To prove this theorem we need to introduce some lemmata on statistical distances. We let $\Delta_{SD}(D_1, D_2)$ denote the standard statistical distance between two distributions D_1 and D_2 which are defined over a common domain X, i.e.

$$\Delta_{SD}(D_1, D_2) = \frac{1}{2} \sum_{x \in X} |D_1(x) - D_2(x)|.$$

Security of our usage of *PRSS-Mask.Next* will rely on the following Lemma's, all of which are variants of the following standard Smudging Lemma (see for example Lemma 2.1 of [AJW11]).

Lemma 15 (Standard Smudging Lemma). *Let* $e \in \mathbb{Z}$ *and* $B, m \in \mathbb{N}$ *denote fixed integers, then we have*

$$\Delta_{SD}\left((e + U(-B, B))^m , U(-B, B)^m\right) \le \frac{m \cdot |e|}{B},$$

where $U(-B, B)$ *is the uniform distribution on the integer interval* $[-B, \ldots, B]$.

From the data processing inequality, which says that the statistical distance between two distributions cannot increase by applying any (possibly randomized) function to them, one can immediately deduce

Lemma 16. *Let* $e \in \mathbb{Z}$ *and* $B, m, v \in \mathbb{N}$ *denote fixed integers, then we have*

$$\Delta_{SD}\left((e + \sum_{i=1}^{v} U(-B, B))^m , \sum_{i=1}^{v} U(-B, B)^m\right) \le \frac{m \cdot |e|}{B},$$

where $U(-B, B)$ *is the uniform distribution on the integer interval* $[-B, \ldots, B]$.

However, a more accurate estimation, when $v \ge 2$, can be given by Lemma 18 (see below), which follows, via the data processing inequality, from the following Lemma, whose proof is given in the full version of [DDE$^+$23], which we reproduce here for completeness.

Lemma 17. *Let* $e \in \mathbb{Z}$ *and* $B, m \in \mathbb{N}$ *denote fixed integers, and let* $\mathcal{P} = U(-B, B) + U(-B, B)$. *Then*

$$\Delta_{SD}(\mathcal{P}^m, (e + \mathcal{P})^m) \le \frac{m \cdot |e|}{B^2} + \sqrt{m \cdot \frac{|e|^2 \cdot \log B + 2}{2 \cdot (B^2 + B)}}.$$

Proof. In the following, we show the case of $e = 1$. The general case follows from the triangle inequality and $|e|$ applications of the lemma for $e = 1$.

Let $b = 2 \cdot B + 1$ and note that $\mathcal{P}(x) = (b - |x|)/b^2$ for all $x \in [-2 \cdot B, 2 \cdot B]$. We begin by introducing two truncated distributions, $\mathcal{P}_\perp$ and $\mathcal{P}_\top$. Define $\mathcal{P}_\perp$ such that it is proportional to $\mathcal{P}$, except that $\mathcal{P}_\perp(-2 \cdot B) = 0$, i.e. we have $\mathcal{P}_\perp(x) = S \cdot (b - |x|)/b^2$ for all $x \in [-2 \cdot B + 1, 2 \cdot B]$, where $S = b^2/(b^2 - 1)$ is the normalization factor. Similarly, we define $\mathcal{P}_\top$ to be proportional to $1 + \mathcal{P}$ but truncated at the top such that we have $\mathcal{P}_\top(2 \cdot B + 1) = 0$. Note that, the normalization factor of $\mathcal{P}_\top$ matches the one of $\mathcal{P}_\perp$ and that the two distributions have the same support. We will show the result by bounding $\Delta_{SD}(\mathcal{P}^m, \mathcal{P}_\perp^m), \Delta_{SD}(\mathcal{P}_\perp^m, \mathcal{P}_\top^m)$, and $\Delta_{SD}(\mathcal{P}_\top^m, (1 + \mathcal{P})^m)$. The rest follows by the triangle inequality.

First note that $\Delta_{SD}(\mathcal{P}, \mathcal{P}_\perp) = \Delta_{SD}(\mathcal{P}_\top, 1 + \mathcal{P}) = 1/b^2$. Accordingly, we have $\Delta_{SD}(\mathcal{P}^m, \mathcal{P}_\perp^m) \le m/b^2$ and $\Delta_{SD}(\mathcal{P}_\top^m, (1 + \mathcal{P})^m) \le m/b^2$.

It remains to bound $\Delta_{SD}(\mathcal{P}_\perp^m, \mathcal{P}_\top^m)$. We first consider the KL–divergence between $\mathcal{P}_\perp$ and $\mathcal{P}_\top$:

$$\Delta_{KL}(\mathcal{P}_\perp, \mathcal{P}_\top) = - \sum_{x=-2 \cdot B + 1}^{2 \cdot B} \mathcal{P}_\perp(x) \cdot \log \frac{\mathcal{P}_\top(x)}{\mathcal{P}_\perp(x)}$$

$$= -S \cdot \sum_{x=-2 \cdot B + 1}^{2 \cdot B} \mathcal{P}(x) \cdot \log \frac{\mathcal{P}(x - 1)}{\mathcal{P}(x)}$$

$$= -S \cdot \left[\left(\mathcal{P}(0) \cdot \log \frac{\mathcal{P}(-1)}{\mathcal{P}(0)} \right) + \left(\mathcal{P}(2 \cdot B) \cdot \log \frac{\mathcal{P}(2 \cdot B - 1)}{\mathcal{P}(2 \cdot B)} \right) \right.$$

$$\left. + \sum_{x=-2 \cdot B + 1}^{-1} \mathcal{P}(x) \cdot \left(\log \frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} + \log \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right) \right]$$

Note that, we have

$$\mathcal{P}(2 \cdot B) \cdot \log \frac{\mathcal{P}(2 \cdot B - 1)}{\mathcal{P}(2 \cdot B)} = \frac{\log 2}{b^2} \geq 0$$

so this term may be ignored (due to the negative sign of the expression). In the following, we make use of the fact that $\log(1 - 1/x) \geq -2/x$ for all $x \geq 2$. Then we have

$$\mathcal{P}(0) \cdot \log \frac{\mathcal{P}(-1)}{\mathcal{P}(0)} = \frac{1}{b} \cdot \log \frac{b-1}{b} = \frac{1}{b} \cdot \log(1 - 1/b) \geq -2/b^2 \ .$$

Similarly, we have for all $x \in [-2 \cdot B + 1, -1]$

$$\mathcal{P}(x) \cdot \left(\log \frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} + \log \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right)$$

$$= \mathcal{P}(x) \cdot \left(\log \left(\frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right) \right)$$

$$= \frac{b+x}{b^2} \cdot \log \left(\frac{(b+x-1)(b+x+1)}{(b+x)^2} \right)$$

$$= \frac{b+x}{b^2} \cdot \log \left(\frac{(b+x)^2 - 1}{(b+x)^2} \right)$$

$$= \frac{b+x}{b^2} \cdot \log \left(1 - \frac{1}{(b+x)^2} \right)$$

$$\geq -\frac{2}{b^2 \cdot (b+x)} \ .$$

Combined, we get

$$\Delta_{KL}(\mathcal{P}_\perp, \mathcal{P}_\top) \leq S \cdot \left[\frac{2}{b^2} + \sum_{x=-2 \cdot B + 1}^{-1} \frac{2}{b^2 \cdot (b+x)} \right]$$

$$= \frac{2 \cdot S}{b^2} \cdot \left(1 + \sum_{x=2}^{2 \cdot B} 1/x \right)$$

$$= \frac{2 \cdot S}{b^2} \cdot \sum_{x=1}^{2 \cdot B} 1/x$$

$$\leq \frac{2 \cdot (\log(2 \cdot B) + 1)}{(b^2 - 1)}$$

$$= \frac{\log B + 2}{2 \cdot (B^2 + B)}.$$

By the sub–additive property of Δ_{KL} we now have

$$\Delta_{KL}(\mathcal{P}_\perp^m, \mathcal{P}_\top^m) \leq m \cdot \frac{\log B + 2}{2 \cdot (B^2 + B)}$$

and by Pinsker's inequality

$$\Delta_{SD}(\mathcal{P}_\perp^m, \mathcal{P}_\top^m) \leq \sqrt{\Delta_{KL}(\mathcal{P}_\perp^m, \mathcal{P}_\top^m)} = \sqrt{m \cdot \frac{\log B + 2}{2 \cdot (B^2 + B)}}.$$

$\square$

Lemma 18. *Let $e \in \mathbb{Z}$ and $B, m, v \in \mathbb{N}$ denote fixed integers with $v \geq 2$, then we have,*

$$\Delta_{SD}\Big((e + \sum_{i=1}^{v} U(-B, B))^m, \ \sum_{i=1}^{v} U(-B, B)^m\Big)$$

$$\leq \frac{m \cdot |e|}{B^2} + \sqrt{m \cdot \frac{|e|^2 \cdot \log B + 2}{2 \cdot (B^2 + B)}},$$

where $U(-B, B)$ is the uniform distribution on the integer interval $[-B, \ldots, B]$.

In our application to proving Theorem 6 we always utilize $v \geq 2$. From which the proof of Theorem 6 then follows from applying Lemma 18. In our application we will set $B = 2^{stat} \cdot |e|$, where $stat = 40$, since Lemma 18 tells us that distinguishing the two distributions (for fixed e) requires around

$$\frac{B^2}{|e|^2 \cdot \log B} = \frac{2^{2 \cdot stat} \cdot |e|^2}{|e|^2 \cdot (stat + \log |e|)}$$

$$= \frac{2^{2 \cdot stat}}{stat + \log |e|} \approx 2^{2 \cdot stat}$$

samples.

Later, when we apply this (say) for m distributed decryption queries we are actually sampling a different value of e per query. On each application, the specific e value used is the output noise term from a bootstrapping operation for a given input ciphertext. Thus the above distances are simplified, upper bounds in our application scenario of the actual statistical distances between the various distributions we analyze.

8.4.1.3   MPC Algorithms: As explained in Section 7.4.1.3, on top of the basic MPC functionality of Figure 109 (possibly extended with the command in Figure 110) we execute "programs". These "programs" consist of sequences of additions and multiplications of secret shared values, the generation of random secret sharings, and the opening of secret sharings to various subsets of parties. Thus, these programs consist of parties executing the operations on the functionality in Figure 109. As argued in [BDO+21] the security of running these programs (within the MPC engine) is related to whether the "trace" of the program execution can itself be simulated. With the security of the program being related to whether the said simulation is perfect, or just statistically secure. This formalizes arguments that have been prior used to argue about the security of MPC algorithms.

As explained in Section 7.4.1.3 one can treat the algorithms to generate secret shared random bits as part of the application of the MPC engine, or we can add them as commands to the MPC functionality itself. It is convenient for presentation purposes to treat them as parts of Layer Three, i.e. they are given in Figure 67, thus we need to treat them as part of the MPC functionality. Hence, to the MPC functionality we also add the additional command *GenBit* to the ideal functionality $\mathcal{F}_{MPC}$, as described in Figure 111. We call the resulting functionality $\mathcal{F}_{MPC^+}$.

8.4.1.4   Bit Generation when $q = 2^k$:   Our protocol, given in Figure 67, to implement the ideal functionality command *GenBit* in the case of $q = 2^k$ follows from the method of [OSV20], extended to the Galois Ring setting from the finite field setting. That this method is correct is trivial, and that it is secure in the $\mathcal{F}_{MPC}$–hybrid model follows from the fact that $\mathcal{F}_{MPC}$ implements an ABB over the Galois Ring.

GenBit Command

> *GenBit*: On receiving (*GenBit*, $varid_a$) from all parties
>   1. The functionality samples a value $a \in \{0, 1\}$ and assigns it to $varid_a$.

Figure 111: Additional Command for Functionality $\mathcal{F}_{MPC}$. This command is only added when $q = 2^k$ or in threshold profile *nSmall*.

8.4.1.5 Bit Generation when $q = p_1 \cdots p_k$: When $q = p_1 \cdots p_k$ we only provide a protocol, in Figure 67, to implement the ideal functionality command *GenBit* in the case of threshold profile *nSmall*. This is because we require access to the additional ideal functionality command *Mask*, as well as the basic ABB operations, and *Mask* is only available in threshold profile *nSmall*. The method (in Figure 67) combines the classical method of [DKL+13] for a large prime, given in Section 7.1.5.3, with a methodology to translate this to a shared bit amongst the product of primes. This extension method is new, but relatively trivial. We prove the following theorem:

Theorem 7. *The bit–generation method* MPC.GenBits(*v*) *for threshold profile nSmall (i.e.* $\binom{\mathfrak{n}}{\mathfrak{t}} < nSmallBnd$) *and* $q = p_1 \cdots p_k$, *given in Figure 67, is both correct and statistically secure.*

Proof. Let p_L denote the largest prime dividing q, by Parameter Choice 2, we can assume that

$$p_L > nSmallBnd \cdot 2^{stat+3} > 8 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat}.$$

The value of $\langle b \rangle$ in line 1(c)vii of method *GenBits* of Figure 67 is clearly a shared bit $\mathfrak{b} \in \{0, 1\}$ modulo p_L; since the generation of this value of $\langle b \rangle$ is just the "in–the–clear" bit generation method for prime fields mapped to the secret shared domain.

The problem is that $\langle b \rangle$ is not a bit modulo q. We have

$$b = \mathfrak{b} + p_L \cdot \lambda$$

for some integer λ. We thus want to remove multiples of p_L.

As we are using the *PRSS-Mask.Next*(1, *stat*) operation shifted by $2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat}$ the unknown value r which we add on to b in line 1(c)ix is generated as a random positive value bounded by

$$r < 4 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot 2^{stat} < p_L/2.$$

In particular $0 \le r < p_L/2$, and hence $\mathfrak{b} + r$ does not wrap around modulo p_L. This means that $b \mod p_L = \mathfrak{b} + r$, thus subtracting the sharing of r as we do in line 1(c)xi gives us a share of $\mathfrak{b}$. This proves correctness of the method for bit generation when $q = p_1 \cdots p_k$.

The final question is whether the value opened in line 1(c)ix reveals any information. By Theorem 6 we have that only after performing this operation $2^{2 \cdot stat}$ times, for the same value of $\mathfrak{b}$, would the adversary be able to distinguish $c = \mathfrak{b} + r$ from the value r. Since, we only ever execute this once per value of $\mathfrak{b}$, clearly the methodology statistically leaks no information. □

8.5 Layer Four

We have shown that our core MPC protocols implement the ideal functionality Figure 109, which itself implements what is known in the literature as an Arithmetic Black Box (ABB). We augmented this with a *Mask* command in Figure 110. This functionality we denoted by $\mathcal{F}_{MPC}$.

We also augmented it with the *GenBit* command in Figure 111, and also showed that this was secure. Recall we denoted this extended ideal functionality by $\mathcal{F}_{MPC^+}$. Thus any algorithm which only uses

the commands of the arithmetic black box (augmented with the *Mask* and *GenBit* command) will be securely implemented by the MPC functionality; i.e. the only information that can leak from the execution of the algorithm is that which can be determined from the algorithms outputs.

The goal of Layer Four of our protocol stack is to implement two ideal functionalities. The first $\mathcal{F}_{KeyGen}$, in Figure 112, acts as a set-up assumption for our protocol, needed for the UC proof we provide. It generates a key pair, and secret shares the secret key among the players using the secret sharing scheme. The modulus q for the secret sharing of the key is equal to Q_1 in the case of BGV and BFV, is equal to $\overline{Q}$ in the case of TFHE when *TFHE.SwitchSquash* is used, and is equal to Q in the case of TFHE when the high round threshold decryption protocol is used. Note, despite wanting active security we do not "complete" adversarially input shares into a complete sharing (as is often done in such situations), as the implementing robustly secure protocol for $\mathcal{F}_{KeyGen}$ does not actually need to do this.

$\mathcal{F}_{KeyGen}$

Init():
    1. Execute $(\mathfrak{pk}, \mathbf{s}) \leftarrow KeyGen(1^{\kappa})$ for the underlying FHE scheme.
    2. Generate a secret sharing $\langle \mathbf{s} \rangle$ of the secret key. using algorithm *Share*($\mathbf{s}$) from Figure 34.
    3. Send $\mathfrak{pk}$ to all players (including the adversary), and send $\langle \mathbf{s} \rangle_i$ to player $\mathcal{P}_i$ (including adversarially controlled players).
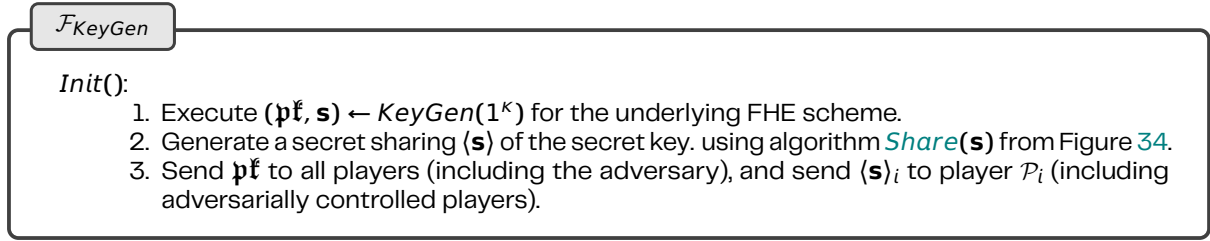
Figure 112: The Ideal Functionality for Distributed Key Generation.

The second functionality we want to implement is $\mathcal{F}_{KeyGenDec}$, given in Figure 113. Note that, this functionality always returns the correct result, irrespective of what the adversary does.

$\mathcal{F}_{KeyGenDec}$

Init():
    1. Execute $(\mathfrak{pk}, \mathbf{s}) \leftarrow KeyGen(1^{\kappa})$ for the underlying FHE scheme.
    2. Send $\mathfrak{pk}$ to all players, including the adversary and store the value $\mathbf{s}$.

Threshold-Dec($\mathfrak{ct}, \mathcal{U}$): For a "decryptable" ciphertext $\mathfrak{ct}$.
    1. Compute $m \leftarrow Dec(\mathfrak{ct}, \mathbf{s})$.
    2. If $\mathcal{U}$ contains any adversarially controlled players then send $(\mathfrak{ct}, m)$ to the adversary.
    3. Otherwise send m to the players in $\mathcal{U}$ and $\mathfrak{ct}$ to the adversary.
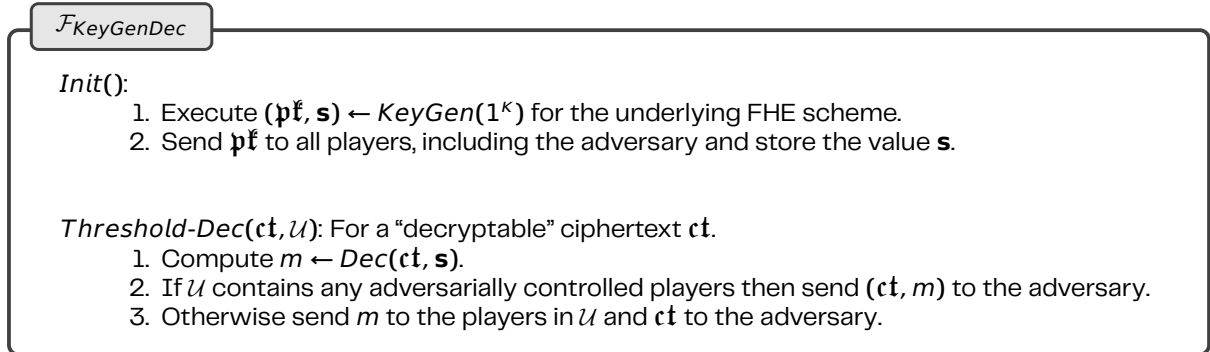
Figure 113: The Ideal Functionality for Distributed Key Generation and Decryption.

8.5.1 Distributions

Since the protocols *MPC.NewHope*(N, B) and *MPC.TUniform*($N, -2^b, 2^b$) in Figure 75 utilize only commands from the functionality $\mathcal{F}_{MPC^+}$, and they implement exactly the same algorithms as one would do to sample from these distributions in the clear, we can conclude that the protocols are both correct and secure.

8.5.2 BGV

8.5.2.1 Threshold Key Generation: The protocol *BGV.Threshold-KeyGen*(N, Q, P, B, R), i.e. thresh–old key generation protocol for BGV, given in Figure 76, is the execution (over the augmented ABB) of the in–the–clear operations one would execute to implement the key generation in the clear; i.e.

BGV.KeyGen(N, Q, P, B, R) from Figure 8. The only difference being that the secret values are secret shared as opposed to being operated on directly. Thus the algorithm is simply the execution of an arithmetic circuit, combined with a method to securely generate secret shared random bits from the uniform distribution on $\{0, 1\}$. Thus, trivially, our protocol for BGV key generation securely realize the functionality $\mathcal{F}_{KeyGen}$ in the $\mathcal{F}_{MPC^+}$–hybrid model.

8.5.2.2 Threshold Decryption: See the analysis for *TFHE.Threshold-Dec*-1, the TFHE threshold decryption method, below.

8.5.3 BFV

8.5.3.1 Threshold Key Generation: Again, the same argument as applied above re threshold BGV key generation also applies here. Thus we can easily conclude that our protocol for threshold BFV key generation is both correct and secure.

8.5.3.2 Threshold Decryption: See the analysis for *TFHE.Threshold-Dec*-1, the TFHE threshold decryption method, below.

8.5.4 TFHE

8.5.4.1 Threshold Key Generation: Again, the same argument as applied above re threshold BGV and threshold BFV key generation also applies here. Thus we can, again, easily conclude that our protocol for threshold TFHE key generation is both correct and secure.

8.5.4.2 Threshold Decryption Method 1:   Recall, for TFHE we have two forms of threshold decryption, which are controlled by the variable $\overline{flag}$. When $\overline{flag} = true$ we utilize the method based on extending the modulus from Q to $\overline{Q}$ and using noise flooding, i.e. *TFHE.Threshold-Dec*-1 from Figure 82. In threshold profile *nSmall* this protocol we will show to be secure in the $\{\mathcal{F}_{KeyGen}, \mathcal{F}_{MPC}\}$–hybrid model. The same argument applies to the protocols *BGV.Threshold-Dec* and *BFV.Threshold-Dec* so we do them all together.

Before doing so we note that in threshold profile *nLarge* the protocols security (in the case of TFHE threshold decryption), follows from basically the same reason for being secure as that given below. Indeed the analysis is slightly simpler as we do not have to worry about *PRSS-Mask.Next* evaluations, and simply need to simulate uniformly random masks pulled from the distribution $TUniform(1, -2^b, 2^b)$ for $b = stat + \lceil \log_2 B_{SwitchSquash} \rceil$. We thus leave the simplifications to the following discussion to the reader to work out.

Our threshold decryption operation is not quite the same as just evaluating the in–the–clear method for decryption in secret shared form, and then revealing the answer. In these cases we first apply noise flooding and then we open the result, to obtain a value which is sufficiently random, but still encodes the message.

We thus need to demonstrate that the three protocols *BGV.Threshold-Dec*, *BFV.Threshold-Dec* and *TFHE.Threshold-Dec*-1 reveal no information about the underlying message, i.e. that the noise flooding applied is large enough to mask the information encoded in the noise term, yet also small enough to not return an invalid decryption value. We discuss the case for TFHE, with the analysis for BGV and BFV following in the same manner. The case for TFHE is slightly more complex in any case due to the need to apply the *TFHE.SwitchSquash* operation in order to produce enough "room" to apply noise flooding.

Correctness for *TFHE.Threshold-Dec*-1 was discussed in Section 7.5.4.2 where we noted that the pa–rameters are chosen to enable the output of the *TFHE.SwitchSquash* operation to produce a ciphertext

$\overline{\mathfrak{ct}} = (\overline{\mathbf{a}}, \overline{b}, \sigma_{\overline{\mathfrak{ct}}})$ such that

$$\|p\| = \|\overline{b} - \overline{\mathbf{a}} \cdot \overline{\mathbf{s}}\| \le B_{SwitchSquash}$$

where

$$2 \cdot nSmallBnd \cdot 2^{stat} \cdot B_{SwitchSquash} < \frac{\overline{\Delta}}{2}.$$

To demonstrate security we provide a simulator $Sim_{TFHE.Threshold\text{-}Dec}$ which simulates the output of the algorithm $TFHE.Threshold\text{-}Dec\text{-}1$ given only access to the functionality $\mathcal{F}_{KeyGenDec}$. Having access to this functionality means the simulator has access to the message which is encrypted by the input ciphertext $\mathfrak{ct}$. The simulator works in the $\{\mathcal{F}_{KeyGen}, \mathcal{F}_{MPC}\}$–hybrid model; thus the simulator emulates these functionalities, and hence the simulator has access to the secret keys of the *dishonest* parties, as well as the keys r_A used for the $PRSS\text{-}Mask.Next$ evaluation of the *dishonest* parties.

Simulator Threshold Decryption

$Sim_{TFHE.Threshold\text{-}Dec}(\mathfrak{ct}, PK, m, I, \{\langle \overline{\mathbf{s}} \rangle\}_{i \in I}, \{r_A\}_{A \cap I \ne \varnothing}, cnt_{Mask})$:
   On input of
   - A ciphertext $\mathfrak{ct} = (\mathbf{a}, b)$ and a public key PK.
   - The underlying message m encrypted by $\mathfrak{ct}$.
   - A set of adversarial parties I with $|I| \le t$.
   - The share values $\langle \overline{\mathbf{s}} \rangle_i$ for $i \in I$.
   - The PRSS secret keys r_A for all subsets $A \subset \{1, \dots, \mathfrak{n}\}$ of size $\mathfrak{n} - t$ such that $A \cap I \ne \varnothing$.
   - The counter cnt_{Mask} used by the $PRSS\text{-}Mask.Next$ algorithm for this specific execution.
   this algorithm outputs the simulated shares $\{\langle c \rangle_j\}_{j \notin I}$ which the honest parties respond with in the simulated protocol execution.

   1. $\overline{\mathfrak{ct}} = (\overline{\mathbf{a}}, \overline{b}) \leftarrow TFHE.SwitchSquash(\mathfrak{ct}, \overline{BK})$.
   2. $Bd_1 \leftarrow 2^{stat} \cdot B_{SwitchSquash} = 2^{stat} \cdot c_{err,1} \cdot \sigma_{\overline{\mathfrak{ct}}}$.
   3. The simulator computes, for $i \in I$, the shares of $\langle c \rangle$ which the dishonest parties *should* compute if they followed the protocol. These are given by

$$\langle c \rangle_i = \overline{b} - \overline{\mathbf{a}} \cdot \langle \overline{\mathbf{s}} \rangle_i + \sum_{A:i \in A} x \Big(\phi^{Bd_1}(r_A, cnt_{Mask}) + \phi^{Bd_1}(r_A, cnt_{Mask} + 1) \Big) \cdot f_A(\alpha_i),$$

   where the sum is over all the sets A above such that $i \in A$.
   4. The simulator computes a simulated value for the flooding term added on

$$t = \sum_{A:A \cap I \ne \varnothing} \Big(\phi^{Bd_1}(r_A, cnt_{Mask}) + \phi^{Bd_1}(r_A, cnt_{Mask} + 1) \Big) \quad + \sum_{B:B \cap I = \varnothing} (r_B + r'_B)$$

   where r_B and r'_B are chosen uniformly at random so that $|r_B|, |r'_B| \le Bd_1$, and the first sum is over all the subsets A of size $\mathfrak{n} - t$ which have a non–trivial intersection with I, and the second sum is over all subsets of size $\mathfrak{n} - t$ which have no intersection with the set I.
   5. The simulator computes $c = \overline{\Delta} \cdot m + t$.
   6. The simulator generates the decryption shares $\{\langle c \rangle_j\}_{j \notin I}$ via Lagrange interpolation (and possibly generating random shares if $|I| < t$) from c and the values $\{\langle c \rangle_i\}_{i \in I}$.
   7. The simulator outputs $\{\langle c \rangle_j\}_{j \notin I}$.

Figure 114: Simulator for $TFHE.Threshold\text{-}Dec\text{-}1(\mathfrak{ct} = (\mathbf{a}, b), PK, \langle \overline{\mathbf{s}} \rangle)$..

We present the heart of the simulator in Figure 114. The method in Figure 114 produces the simulated shares (turning this into a simulator for the full threshold decryption protocol is easy left to the reader). The simulator essentially outputs a simulator value c_{Sim}, which has a different distribution from the value c_{Real} which a genuine execution will follow. In particular c_{Real} will be of the form

$$\overline{\Delta} \cdot m + e + t$$

whereas c_{Sim} will be of the form

$$\overline{\Delta} \cdot m + t.$$

The distinguisher after threshold decryption knows the value of m, and so the task of the distinguisher is essentially to distinguish samples of the form $e + t$ from samples of the form t, where e is a value bounded by $B_{SwitchSquash}$.

In line 4 the simulator contributes at least two uniformly random values in the range $[-2^{stat} \cdot Bd_1, \ldots, 2^{stat} \cdot Bd_1]$ to the sum for t, which are unknown to the adversary, and hence unknown to the distinguisher. Lemma 18 tells us that to have any advantage in such a distinguishing game the adversary would need to obtain at least $2^{2 \cdot stat}$ such samples, for the same value of e. In particular this would require at least $2^{2 \cdot stat}$ calls to the threshold decryption operation for the same input ciphertext $\mathfrak{ct}$.

8.5.4.3   Threshold Decryption Method 2:   When $\overline{flag} = false$ we use a threshold decryption method based on bit–decomposition modulo Q, i.e. *TFHE.Threshold-Dec-2* from Figure 85. This protocol can be shown to be secure in the $\{\mathcal{F}_{KeyGen}, \mathcal{F}_{MPC^+}\}$–hybrid model.

The bit–manipulation methods of Figure 83 and Figure 84 are all translations of the standard in–the–clear bit–manipulation algorithms into the MPC domain, and so by security of the underlying ABB they are correct and secure. The threshold decryption algorithm *TFHE.Threshold-Dec-2* is (essentially) the translation of the in–the–clear algorithm (partial decryption followed by message extraction via truncation), albeit in a slightly round–about way. Since the bit–manipulation methods are all secure it is clear that the algorithm realizes the functionality $\mathcal{F}_{KeyGenDec}$ in the $\{\mathcal{F}_{KeyGen}, \mathcal{F}_{MPC^+}\}$–hybrid model.

8.5.5   Resharing

The method given in Figure 86 for robustly resharing a secret sharing from one set of parties S_1 to another set of parties S_2 is folklore, and can be seen alluded to in the literature dating back to the late 1980s. The idea is simply one of masking, computing the syndrome polynomial in secret shared form, opening the syndrome polynomial and then applying syndrome decoding (in the clear) in order to detect error locations.

8.6   Layer Five

8.6.1   Interpreting Encryption as a Subset Sum

There is nothing to discuss about this section in relation to security or analysis.

8.6.2   Pairing Based Elliptic Curves

If sec is the security parameter, we need at least $l(sec) > 2 \cdot sec$ (due to Pollard's rho algorithm) but the specific l depends on the underlying elliptic curve and on the application. Indeed, $\mathbb{G}_T$ is a subgroup of order r in $\mathbb{F}_{t^k}^*$, where k is the smallest integer such that $r | t^k - 1$ if $\mathbb{F}_t$ is the base field of the curve $E(\mathbb{F}_t)$ containing $\mathbb{G}$. So, the extension field $\mathbb{F}_{t^k}^*$ should be large enough to make sure that computing discrete logarithms in $\mathbb{F}_{t^k}^*$ is infeasible. Moreover, to guarantee the hardness of computing discrete logarithms α when many elements $\{g^{(\alpha^i)}\}_{i=1}^{\bar{B}}$ are given, we need to slightly increase the group order (i.e., by $\log \bar{B}$ bits) to hedge against Cheon's attack [Che06].

If a BLS12 curve (where $k = 12$) is used, r and t can be generated from the parameterization

$$r = u^4 - u^2 + 1$$
$$t = (u^6 - 2 \cdot u^5 + 2 \cdot u^3 - 2u + 1)/3 + u$$

by tuning the seed u (see, e.g., [Sco19]). The BLS12–381 curve[20] (for which $r \approx 2^{255}$) is widely used but is believed to provide slightly less than 128 bits of security (between 117 and 120 bits). To obtain 128 bits of security, one way use the BLS12–446 curve[21] which yields a group order $r \approx 2^{299}$ (thus providing a sufficient margin against Cheon's algorithm) such that $r - 1$ is divisible by a large power of 2 (which allows large–dimension FFTs in $\mathbb{Z}/(r)$). This curve is obtained from the seed $u = -(2^{74} + 2^{73} + 2^{63} + 2^{57} + 2^{50} + 2^{17} + 1)$.

8.6.3 ZKPoKs Based on Vector Commitments

8.6.3.1 Required Hash Functions: The security of the hash functions reduces to the security of the underlying *XOF* construction.

8.6.3.2 CRS and the CRS SetUp: The protocols in Figure 88 and Figure 89 are designed in such a way that the NIZK and soundness properties are preserved as long as at least one of the K participants is honest. Due to their sequential nature, they cannot guarantee that the resulting product $\alpha = \prod_{i=1}^{K} \tau_i$ is uniformly distributed. Indeed, a rushing adversary can choose α_K as a function of $g^{\prod_{i=1}^{K-1} \tau_i}$ and force a few bits of $g^{\prod_{i=1}^{K} \tau_i}$ to be zero.

Despite this bias, [KMSV21] showed that ceremonial versions of [Gro16], for example, can still be proven secure as long as one of the contributors behaves honestly (meaning that it randomly chooses its contribution τ_i and erases it after the ceremony protocol). For security reasons, at each round i, the i–th participant has to demonstrate knowledge of its randomizer τ_i. In [KMSV21], this was done using BLS–type [BLS01] proofs–of–possessions [RY07] that were shown to satisfy a suitable notion of simulation–extractability. Nikolaenko *et al.* [NRBB24] used Schnorr–type [Sch90] Σ–protocols in order to optimize the consistency checks on the CRS updates after each round. We recall the ceremony protocol of Nikolaenko *et al.* [NRBB24] in Figure 89.

If the j–th contributor is honest and generates a well–formed pp_j, then (35) is satisfied because

$$
e\Big(\prod_{\substack{i=1 \\ i \neq \bar{B}_1+1 \\ i \neq \bar{B}_1+2}}^{2\bar{B}_1} g_{i,j}^{(\rho_1^{i-1})}, \ \hat{g} \cdot \prod_{\ell=1}^{\bar{B}_1-1} \hat{g}_{\ell,j}^{(\rho_2^\ell)} \Big) = e\Big(g_{1,j} \cdot \prod_{\substack{i=2 \\ i \neq \bar{B}_1+1 \\ i \neq \bar{B}_1+2}}^{2\bar{B}_1} g_{i,j}^{(\rho_1^{i-1})}, \ \hat{g} \cdot \prod_{\ell=1}^{\bar{B}_1-1} \hat{g}_{\ell,j}^{(\rho_2^\ell)} \Big)
$$

$$
= e\Big(g_{1,j} \cdot \prod_{\substack{i=1 \\ i \neq \bar{B}_1 \\ i \neq \bar{B}_1+1}}^{2\bar{B}_1-1} g_{i+1,j}^{(\rho_1^i)}, \ \hat{g} \cdot \prod_{\ell=1}^{n-1} \hat{g}_{\ell,j}^{(\rho_2^\ell)} \Big)
$$

$$
= e\Big(g \cdot \prod_{\substack{i=1 \\ i \neq \bar{B}_1 \\ i \neq \bar{B}_1+1}}^{2\bar{B}_1-1} g_{i,j}^{(\rho_1^i)}, \ \hat{g}_{1,j} \cdot \prod_{\ell=1}^{\bar{B}_1-1} \hat{g}_{\ell+1,j}^{(\rho_2^\ell)} \Big)
$$

$$
= e\Big(g \cdot \prod_{\substack{i=1 \\ i \neq \bar{B}_1 \\ i \neq \bar{B}_1+1}}^{2\bar{B}_1-1} g_{i,j}^{(\rho_1^i)}, \ \hat{g}_{1,j} \cdot \prod_{\ell=2}^{\bar{B}_1} \hat{g}_{\ell,j}^{(\rho_2^{\ell-1})} \Big)
$$

$$
= e\Big(g \cdot \prod_{\substack{i=1 \\ i \neq \bar{B}_1 \\ i \neq \bar{B}_1+1}}^{2\bar{B}_1-1} g_{i,j}^{(\rho_1^i)}, \ \prod_{\ell=1}^{\bar{B}_1} \hat{g}_{\ell,j}^{(\rho_2^{\ell-1})} \Big)
$$

---

[20] https://neuromancer.sk/std/bls/BLS12-381
[21] https://neuromancer.sk/std/bls/BLS12-446.

The second equality of (35) can be verified in the same way.

8.6.3.3 Method 1 Proof Construction:

The soundness of the argument system of Figure 90 and Figure 91 is proven in the algebraic group model [FKL18] and the random oracle model under the assumption that, given

$$(g, \hat{g}, \{g^{(\alpha^i)}\}_{i=1}^{2\bar{B}_1}, \{\hat{g}^{(\alpha^i)}\}_{i=1}^{\bar{B}_1})$$

for a random $\alpha \in \mathbb{Z}/(r)$, computing α is infeasible. In [Lib24, Theorem 6], the construction given in Figure 90–Figure 91 is shown to provide the strong notion of simulation–extractability which preserves knowledge–soundness against malicious provers that can observe simulated proofs and try to modify them in an attempt to create a proof that defeats the knowledge–extractor.

The security proof in [Lib24, Theorem 6] shows that the soundness of the NIZK argument under the assumption that computing α from $(g, \hat{g}, \{g^{(\alpha^i)}\}_{i=1}^{2\bar{B}_1}, \{\hat{g}^{(\alpha^i)}\}_{i=1}^{\bar{B}_1})$ is infeasible. However, the latter assumption assumes that α is uniformly distributed in $\mathbb{Z}/(r)$. As previously mentioned, the ceremony protocol described in Figure 89 does not guarantee the uniformity of the trapdoor $\alpha = \prod_{i=1}^{K} \tau_i$ hidden in the CRS. Fortunately, we can still prove the security of the ceremonial version of the NIZK argument as long as at least one participant is honest during the ceremony. This will be given in the full version of [Lib24] in which it is shown in the combined AGM+ROM model, that, under the $(2\bar{B}_1, \bar{B}_1)$–DLOG assumption, the knowledge extractor can always extract the witness τ_j from the Schnorr proofs $\bar{\pi}_{PoK_j}$ at each round j involving an adversarial contribution to the CRS.

8.6.3.4 Method 2 Proof Construction:

Similarly to the first pairing–based construction given above, the simulation–extractability of the NIZK argument of Figure 93–Figure 96 is proven in the combined algebraic group and random oracle model.

Intuitively, the proof builds a polynomial–time knowledge extractor that uses the algebraic representation of all group elements contained in an adversarially–generated NIZK proof to extract a witness w for the proven statement. Then, the security proof shows that, if the witness w obtained by the knowledge extractor is not a valid witness, the extractor can break the $(2\bar{B}_1, \bar{B}_1)$–DLOG assumption by computing α from the group elements contained in the CRS. This is done by computing α as a root of a polynomial which is non–zero with overwhelming probability if the extracted w is not a valid witness. A detailed proof is given [BEL+24, Theorem 2]. It proceeds in a similar way to the security analysis of the NIZK construction of Method 1 with the difference that it also uses ideas from [LNS21, GHL22] to argue that the knowledge extractor obtains a valid witness.

8.6.4 ZKPoKs Based on MPC–in–the–Head

Security of the MPC–in–the–Head based proofs given in Section 7.6.4 follows the standard analysis introduced in [KKW18]; and the associated extensions given in [FMRV22, AGH+23, FR23]. In this section we present a less formal/intuitive explanation of the security analysis, and the reasoning behind the soundness values from Section 7.6.4. For a more formal analysis, and the relevant theorems we refer to the papers [FMRV22, AGH+23, FR23] from which we take our MPC–in–the–Head protocols.

An MPC–in–the–Head zero–knowledge proof consists of two stages, an offline stage which corresponds to the pre–processing stage of many MPC protocols. In this stage pre–processed "sharings" are produced which are shown with high probability to satisfy a given predicate. In our protocols the offline phase generates secret shared random bits. This offline phase is proven secure using a traditional cut and choose argument, just as in [KKW18]. Note, we do not need an offline phase which produces data to check 'multiplication gates', as the statements we are proving are purely linear statements on the underlying secret values. This is because we are only proving correctness of subset–sum equations, and so the overall MPC–in–the–Head machinery we need is simpler than that required for more general statements.

In the online stage the pre-processed values are consumed, as an MPC protocol is emulated between n virtual parties (we say virtual parties, as the parties live in the provers head and are not actual parties in a real protocol). The views of all virtual parties are committed to by the prover. The virtual parties view consists of the internal randomness and the communication received by the specific virtual party during the simulated protocol execution from the other virtual parties.

To ensure a sound proof system the combined online and offline phases need to be secure. We only fully execute the τ combined offline and online phases, but we execute M offline phases (and by execute we again mean 'in-the-head'). These executions are committed to, via the views mentioned earlier.

To check the offline phase has been executed correctly $M - \tau$ of the offline phases are fully opened and verified. To check the τ online phases are executed correctly a given number of parties' online executions are opened. The precise number of parties opened depends on the threshold of the underlying secret sharing scheme. By opening a view one means revealing the commitment to the previous virtual parties' view. The zero-knowledge property of the final proof relies on the privacy of the underlying secret sharing scheme; thus for full threshold based secret sharing we can open a maximum of $n - 1$ parties, and for Shamir sharing we can open a maximum of t parties.

We select the maximum value to open (subject to our privacy constraint) in order to minimize the soundness error of our proof. This is because the soundness of the online phase is given by the inability of the adversary to ahead of time select which set of parties will not be opened, thus the soundness is (roughly) either $1/n$ or $1/\binom{n}{t}$ (depending on the precise variant one is considering). To obtain a suitably small soundness ratio the protocol needs to be repeated.

To cheat the adversary can also select, ahead of time, c of the offline phases in which it will cheat. Of course we need the number of correct offline executions $M - c$ to be at least the number of opened offline executions $M - \tau$ otherwise the adversary is always caught (so $c \leq \tau$). Then, on the assumption that the adversary passes the offline check, there are c out of the τ executions which are already corrupted, for those the adversary does not have to do anything else. But for the remaining $\tau - c$ unbiased executions the adversary has to cheat, and for these it has to guess which party is not going to be opened.

The adversary wants to maximize its probability of cheating. Let c be the number of offline executions out of the M in which the prover cheats, equivalently he executes $j = M - c$ of these executions validly. Let A_c denote the event that of the M total offline executions, none of the $M - \tau$ which are opened coincide with the c ones which the adversary cheats on. Since $M - \tau$ executions are totally revealed and checked by the verifier, the success probability of the adversary in the offline phase is

$$\Pr[A_c] = \frac{\binom{M-c}{M-\tau}}{\binom{M}{M-\tau}} = \frac{\binom{j}{M-\tau}}{\binom{M}{M-\tau}}.$$

Thus we must have $c \leq \tau$, i.e. $j \geq M - \tau$, i.e. the adversary has to be lucky enough that all of his cheating occurs in the τ executions processed in the online phase.

On the assumption that the offline phase passes successfully despite the verifier cheating in c of them, we can compute the probability that the online phase passes successfully too. This probability is approximately $p_c = n^{c-\tau}$ or $p_c = \binom{n}{t}^{c-\tau}$, corresponding to the probability that on all of the $\tau - c$ online phases that rely on a correct offline phase, the verifier has correctly guessed which party to cheat in. For the protocols based on full threshold the probability gets modified slightly, as we are simultaneously executing a form of rejection sampling in order to keep the proof size small. Thus we obtain $p_c \approx n^{c-\tau}$ only when $\eta = 0$ in the full threshold proofs. For the protocol based on Shamir sharing this probability is $p_c \approx \binom{n}{t}^{c-\tau}$. The $\approx$ in these values is because we also need to take into account the false positive rate for the statement.

Thus the overall success probability is given by

$$err = \max_{0 \leq c \leq \tau} \left\{ \frac{\binom{M-c}{M-\tau}}{\binom{M}{M-\tau}} \cdot p_c \right\}$$

$$= \max_{M-\tau \leq j \leq M} \left\{ \frac{\binom{j}{M-\tau}}{\binom{M}{M-\tau}} \cdot p_{M-j} \right\}.$$

This leads to the formulae for *err* given below.

8.6.4.1 Method 1: Full Threshold Based Proofs: The statement we are trying to prove has a false positive rate of roughly $1/q^d$, since a random assignment to the input variables will produce a solution with probability $1/q^d$ (as we have d equations modulo q we need to solve). Thus a single execution of the "online" part of the MPC–in–the–Head zero-knowledge proof will have soundness error (if we set $\eta = 0$)

$$\frac{1}{n} + \frac{1}{q^d} \cdot \left(1 - \frac{1}{n} \right).$$

Since in our applications we have $n \ll q$ we will approximate this soundness error by $1/n$. However, a non–zero η will increase this value slightly to the value p_j below, where j is the number of complete executions which are validly executed.

The proof parameters are constrained by the following equations:

$$p_{local\text{-}rej} = \frac{B}{A} < 1,$$

$$p_{global\text{-}rej} = 1 - \sum_{i=0}^{\eta} \binom{\tau}{i} \cdot (1 - p_{local\text{-}rej})^{\tau-i} \cdot p_{local\text{-}rej}^{i},$$

$$p_j = \sum_{i=0}^{\eta} \left[\binom{j-M+\tau}{i} \cdot \left(1 - \frac{1}{n} \right)^{i} \left(\frac{1}{n} \right)^{j-M+\tau-i} \right],$$

$$err = \max_{M-\tau \leq j \leq M} \left\{ \frac{\binom{j}{M-\tau}}{\binom{M}{M-\tau}} \cdot p_j \right\}.$$

We require that $p_{global\text{-}rej}$ is "small", as this is the probability that the prover will not be able to complete a proof and will need to restart the proof process; a value around 0.01 or less for this will be sufficient. We also require that *err* is exponentially small, as it is the soundness error; so a value of $err < 2^{-dist}$ is sufficient. These values need to be selected also to keep the proof size small as well.

8.6.4.2 Method 2: Full Threshold Based Proofs – Hyper–Cube Variant: The analysis of this method is virtually identical to that of Method 1 above.

8.6.4.3 Method 3: Shamir Based Proofs: This method has a slightly different analysis to the first two; although the general strategy remains the same. The statement we are trying to prove has a false positive rate of roughly $1/q^d$, since a random assignment to the input variables will produce a solution with probability $1/q^d$ (as we have d equations modulo q we need to solve). Thus a single execution of the "online" part of the MPC–in–the–Head zero-knowledge proof will have soundness error

$$p_{n,t} = \frac{1}{\binom{n}{t}} + \frac{t \cdot (n-t)}{q^d \cdot (t+1)}.$$

For "large" q (as when proving BGV or BFV encryptions) this will be approximated by the first term. Hence, in this case we use this exact formulae for the soundness error of one execution.

The overall soundness of the entire proof system is then given by

$$err = \max_{M-\tau \leq j \leq M} \left\{ \frac{\binom{j}{M-\tau}}{\binom{M}{M-\tau}} \cdot p_{n,t}^{M-\tau-j} \right\}.$$

9 Analytic complexity

In this section we analyze the various complexities of our protocols and algorithms, and the related constraints on configurable parameters. We examine round complexity, communication complexity and computational complexity. When measuring communication complexity we give the total amount of data sent, whereas when giving computational complexity we present the amount of work per player.

We also measure complexity on the **happy path**, i.e. when no errors are detected. The reason for this is that in practice active deviation from protocols by adversaries are rare, and once one is detected in our protocol they are removed from the protocol (which may have adverse business consequences). Thus we model the complexity in terms of the expected real life execution of the protocols.

The basic high level parameter input to the threshold protocols are the number of players $\mathfrak{n}$, the threshold $\mathfrak{t}$, the MPC modulus $q = p^k$ or $q = p_1 \cdots p_k$ and the security parameter sec. In the case of $q = p_1 \cdots p_k$ we let p denote the smallest prime dividing q and p_L the largest prime dividing q. We aim to express complexity in terms of the basic high level parameters, i.e. $\mathfrak{n}$, q and sec. Note that $k = O(\log q)$ and $\mathfrak{t} = O(\mathfrak{n})$.

To simplify computational complexity we assume that operations modulo q (or modulo $Q/\overline{Q}$ when looking at the complexity of the FHE schemes) take unit time. This is a gross simplification as addition takes time linear in $\log q$, whilst multiplication lies somewhere between $O((\log q)^{1+\epsilon})$ and $O((\log q)^2)$ (depending on the type of implementation one implemented for multiplication modulo q). We also assume that executions of symmetric primitives such as hash functions also take unit time, and that their output length is a linear function of sec (usually sec for application of a block cipher like AES and $2 \cdot sec$ for application of a hash function like SHA–2 or SHA–3).

The first meta–parameter which must then be defined is the Galois Ring degree, $\mathfrak{d}$, which we recall by Parameter Choice 3 and Parameter Choice 4 must satisfy

$$p^{\mathfrak{d}} > \mathfrak{n} \text{ and } q^{\mathfrak{d}} > 2^{sec}.$$

To simplify exposition we assume a naive school–book multiplication algorithm in the Galois Ring, thus operations in the Galois Ring will take time $O(\mathfrak{d}^2)$. When applied to the BGV/BFV system we will have $\mathfrak{d} = 1$ in any case, and for TFHE the size of $\mathfrak{d}$ is $\lceil \log_2 \mathfrak{n} \rceil$ (since $q = 2^{128}$ and $\mathfrak{n} \geq 4$). A ring element can thus be represented in $O(\mathfrak{d} \cdot \log q) = O(\log \mathfrak{n} \cdot \log q)$ bits.

9.1 Conventional Algorithm Complexities

In this subsection we examine the computational complexity of the underlying FHE schemes. As the focus of this document is on threshold implementations we do not examine, or consider, optimizations of the FHE operations themselves which a real system will implement. Thus the complexities in this section should be considered only "first approximations"; we only list them here as we will sometimes need to refer to them when considering the complexity of the threshold protocols below.

When giving the complexities of these algorithms, recall we assume that operations modulo Q take unit time. In addition, for the TFHE algorithms, we do not analyze the *Expand* sub–routines as they are relatively simple.

9.1.1 *BGV.KeyGen*(N, Q, P, B, R), *BGV.Enc*($\mathbf{m}, \mathfrak{pf}$) **and** *BGV.Dec*($\mathfrak{ct}, \mathfrak{sf}$)

Assuming one uses an NTT algorithm for the ring multiplications the complexity of these algorithms will be $O(N \cdot \log N)$; if naive methods are used it will be $O(N^2)$. When using encryption followed by a zero–knowledge proof of correctness it is likely that the $O(N^2)$ algorithm will be used, since it is in this representation that the subset–sum is created for passing into the proof system.

9.1.2 $ModSwitch^{Q \to q}(\mathbf{a}, \mathbf{b})$ and $BGV.Scale(\mathfrak{ct}, \ell')$

These operations require only modular operations on the coefficients of the associated ring elements, and thus are $O(N)$. However, if the inputs are represented in the NTT domain, then we require $O(N \cdot \log N)$ operations.

9.1.3 $BGV.KeySwitch((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_{input})$

The complexity is dominated by the ring multiplications again; which take $O(N \cdot \log N)$, if we assume the modular operations on the coefficient domain takes unit time. However, the base operations in this algorithm are on integers of size at most $R \cdot Q$. Thus the complexity is more akin to

$$O(N \cdot \log N \cdot (\log R)^2)$$

base operations modulo Q.

9.1.4 $BGV.Add(\mathfrak{ct}_a, \mathfrak{ct}_b)$ and $BGV.Mult(\alpha, \mathfrak{ct})$

These operations are linear in nature and so have complexity $O(N)$.

9.1.5 $BGV.Mult(\mathfrak{ct}_a, \mathfrak{ct}_b)$

This operation's complexity is dominated by the $KeySwitch$ operation, and so has complexity $O(N \cdot \log N \cdot (\log R)^2)$.

9.1.6 $BFV.KeyGen(N, Q, P, B, R)$, $BFV.Enc(\mathbf{m}, \mathfrak{pt})$ and $BFV.Dec(\mathfrak{ct}, \mathfrak{st})$

These are essentially identical to the operations for BGV.

9.1.7 $BGV.toBFV(\mathfrak{ct})$ and $BFV.toBGV(\mathfrak{ct}')$

These are both linear operations, applied to the coefficients of the ring elements, and so have complexity $O(N)$.

9.1.8 $Enc^{LWE}(m, \mathbf{s}; \dots)$

This requires $O(\ell)$ operations modulo Q.

9.1.9 $Enc^{GLWE}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); \dots)$

This performs w operations in a ring of dimension N, and hence can be performed in $O(w \cdot N \cdot \log N)$ operations modulo Q.

9.1.10 $Enc^{Lev}(m, \mathbf{s}; \dots)$

This requires ν calls to Enc^{LWE} and hence requires $O(\nu \cdot \ell)$ operations.

9.1.11 $Enc^{GLev}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); \dots)$

This requires ν calls to Enc^{GLWE} and hence requires $O(\nu \cdot w \cdot N \cdot \log N)$ operations; if we assume an FFT implementation is used for the ring multiplication.

9.1.12 $Enc^{GGSW}(\mathbf{m}, (\mathbf{s}_0, \ldots, \mathbf{s}_{w-1}); \ldots)$

This requires $w + 1$ calls to Enc^{GLev} and hence requires $O(\nu \cdot w^2 \cdot N \cdot \log N)$ operations.

9.1.13 $TFHE.KeyGen(\ldots)$

This executes $\hat{\ell} \; Enc^{Lev}$ of dimension ℓ or $w \cdot N$ depending on $type$, $w \cdot N$ executions of Enc^{Lev} of dimension ℓ, and ℓ executions of Enc^{GGSW} with parameters w and N and if $\overline{flag}$ also ℓ executions of Enc^{GGSW} with parameters $\overline{w}$ and $\overline{N}$. Since the $(\overline{w}, \overline{N})$ parameters are much bigger and the generation of the bootstrapping key dominates that of the other keys, the complexity is either $O(\ell \cdot \overline{\nu_{bk}} \cdot \overline{w}^2 \cdot \overline{N} \cdot \log \overline{N})$ if $\overline{flag}$, or $O(\ell \cdot \nu_{bk} \cdot w^2 \cdot N \cdot \log N)$ otherwise.

9.1.14 $TFHE.Decompose(x, Q, \beta, \nu)$

This decomposes an integer into ν parts, thus requiring $O(\nu)$ operations.

9.1.15 $TFHE.DimensionSwitch(\mathfrak{ct}, PKSK)$

The main cost is step 5 in which we sum $\hat{\ell}$ vector by matrix products where the the vector has length ν_{pksk} and integer components and the matrix is $\nu_{pksk} \times d$ having integer coefficients modulo Q. As such the cost is $O(\hat{\ell} \cdot \nu_{pksk} \cdot d)$.

9.1.16 $TFHE.Enc\text{-}Sub(m, \mathfrak{pk})$

The main cost here is a ring multiplication in a ring of dimension ℓ. Thus the complexity (assuming an FFT operation is used) is $O(\ell \cdot \log \ell)$. However, just as for BGV/BFV above it may be the case that it is simpler to use naive methods if the encryption is followed by a zero–knowledge proof of correctness.

9.1.17 $TFHE.Enc(m, \mathfrak{pk})$

This is just $TFHE.Enc\text{-}Sub$ followed by $TFHE.DimensionSwitch$ with the latter dominating the complexity so we require $O(\hat{\ell} \cdot \nu_{pksk} \cdot d)$ operations.

9.1.18 $TFHE.Dec(\mathfrak{ct}, \mathfrak{sk})$

This executes a dot–product of dimension ℓ, and so requires $O(\ell)$ operations.

9.1.19 $TFHE.Add(\mathfrak{ct}_1, \mathfrak{ct}_2)$, $TFHE.ScalarMult(\alpha, \mathfrak{ct})$ **and** $TFHE.ModSwitch(\mathfrak{ct})$

These are both linear operations on a vector of size ℓ, and so require $O(\ell)$ operations.

9.1.20 $TFHE.Flatten(\mathfrak{ct})$

This is just a reorganization of the input data, of size $w \cdot N$, and thus requires $O(w \cdot N)$ operations.

9.1.21 $TFHE.KeySwitch(\mathfrak{ct}, KSK)$

This is an extension of $TFHE.DimensionSwitch$ which for the sake of complexity we can simply replace $\hat{\ell}$ by $w \cdot N$ and d by ℓ. As such that complexity is $O(w \cdot N \cdot \nu_{ksk} \cdot \ell)$.

9.1.22 *TFHE.ExternalProduct*$(\mathfrak{ct}, CT)$

The main complexity is step 6 which involves the sum of w vector–matrix products with elements in $\mathbf{R}_Q$. The vector is of dimension v and the matrix of size $v \times (w+1)$ so the overall complexity is $O(w^2 \cdot v \cdot N \cdot \log N)$ assuming use of the FFT.

9.1.23 *TFHE.BootStrap*$(\mathfrak{ct}, f, BK)$

This executes ℓ *TFHE.ExternalProduct* operations using the bootstrapping key, thus the complexity is $O(\ell \cdot w^2 \cdot v_{bk} \cdot N \cdot \log N)$ if using the FFT.

9.1.24 *TFHE.PBS*$(\mathfrak{ct}, f, BK)$

This executes a *TFHE.BootStrap* and a *TFHE.KeySwitch*, the order depending on the type of the input. As the *TFHE.BootStrap* is the more costly operation, the overall complexity is $O(\ell \cdot w^2 \cdot v_{bk} \cdot N \cdot \log N)$ when using the FFT.

9.1.25 *TFHE.SwitchSquash*$(\mathfrak{ct}, \overline{BK})$

This is similar to a *TFHE.PBS* however the external product operations are done over the larger dimensional ring, however now the operations are modulo $\overline{Q}$ and not modulo Q (which gives a slightly larger implied constant in the complexity estimate). As such, the complexity is $O(\ell \cdot \overline{w}^2 \cdot \overline{v_{bk}} \cdot \overline{N} \cdot \log \overline{N})$ assuming the use of the FFT.

9.1.26 Conventional Algorithm Summary

We summarize the above discussion in the following table:

Algorithm	Computational Complexity
BGV.KeyGen(N, Q, P, B, R)	$O(N \cdot \log N)$
BGV.Enc$(\mathbf{m}, \mathfrak{pk})$	$O(N \cdot \log N) / O(N^2)$
BGV.Dec$(\mathfrak{ct}, \mathfrak{sk})$	$O(N \cdot \log N)$
ModSwitch$^{Q \to q}(\mathbf{a}, \mathbf{b})$	$O(N)$
BGV.Scale$(\mathfrak{ct}, \ell')$	$O(N)$
BGV.KeySwitch$((\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2), \ell, B_{input})$	$O(N \cdot \log N \cdot (\log R)^2)$
BGV.Add$(\mathfrak{ct}_a, \mathfrak{ct}_b)$	$O(N)$
BGV.Mult$(\alpha, \mathfrak{ct})$	$O(N)$
BGV.Mult$(\mathfrak{ct}_a, \mathfrak{ct}_b)$	$O(N \cdot \log N \cdot (\log R)^2)$
BFV.KeyGen(N, Q, P, B, R)	$O(N \cdot \log N)$
BFV.Enc$(\mathbf{m}, \mathfrak{pk})$	$O(N \cdot \log N) / O(N^2)$
BFV.Dec$(\mathfrak{ct}, \mathfrak{sk})$	$O(N \cdot \log N)$
BGV.toBFV$(\mathfrak{ct})$	$O(N)$
BFV.toBGV$(\mathfrak{ct}')$	$O(N)$
Enc$^{LWE}(m, \mathbf{s}; \dots)$	$O(\ell)$
Enc$^{GLWE}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); \dots)$	$O(w \cdot N \cdot \log N)$
Enc$^{Lev}(m, \mathbf{s}; \dots)$	$O(v \cdot \ell)$
Enc$^{GLev}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); \dots)$	$O(v \cdot w \cdot N \cdot \log N)$
Enc$^{GGSW}(\mathbf{m}, (\mathbf{s}_0, \dots, \mathbf{s}_{w-1}); \dots)$	$O(v \cdot w^2 \cdot N \cdot \log N)$
TFHE.KeyGen$(\dots)$	$O(\ell \cdot \overline{v_{bk}} \cdot \overline{w}^2 \cdot \overline{N} \cdot \log \overline{N})$
TFHE.Decompose(x, Q, β, v)	$O(v)$

$TFHE.DimensionSwitch(\mathfrak{ct}, PKSK)$	$O(\hat{\ell} \cdot \nu_{pksk} \cdot d)$
$TFHE.Enc\text{-}Sub(m, \mathfrak{pk})$	$O(\ell \cdot \log \ell) / O(\ell^2)$
$TFHE.Enc(m, \mathfrak{pk}, PKSK)$	$O(\hat{\ell} \cdot \nu_{pksk} \cdot d)$
$TFHE.Dec(\mathfrak{ct}, \mathfrak{sk})$	$O(\ell)$
$TFHE.Add(\mathfrak{ct}_1, \mathfrak{ct}_2)$	$O(\ell)$
$TFHE.ScalarMult(\alpha, \mathfrak{ct})$	$O(\ell)$
$TFHE.ModSwitch(\mathfrak{ct})$	$O(\ell)$
$TFHE.Flatten(\mathfrak{ct})$	$O(w \cdot N)$
$TFHE.KeySwitch(\mathfrak{ct}, KSK)$	$O(w \cdot N \cdot \nu_{ksk} \cdot \ell)$
$TFHE.ExternalProduct(\mathfrak{ct}, CT)$	$O(w^2 \cdot \nu \cdot N \cdot \log N)$
$TFHE.BootStrap(\mathfrak{ct}, f, BK)$	$O(\ell \cdot w^2 \cdot \nu_{bk} \cdot N \cdot \log N)$
$TFHE.PBS(\mathfrak{ct}, f, BK)$	$O(\ell \cdot w^2 \cdot \nu_{bk} \cdot N \cdot \log N)$
$TFHE.SwitchSquash(\mathfrak{ct}, \overline{BK})$	$O(\ell \cdot \overline{w}^2 \cdot \overline{\nu_{bk}} \cdot \overline{N} \cdot \log \overline{N})$

Table 11: Computational Complexity of the Conventional Algorithms for FHE.

9.2 Layer Zero Algorithm Complexities

9.2.1 $BW(\mathbf{c}, e)$

In the worst case this is called with $e = \lfloor (\mathfrak{n} - \mathfrak{d})/2 \rfloor = O(\mathfrak{n})$. The method requires solving e linear equations in $O(\mathfrak{n})$ unknowns in the finite field $\mathbb{F}_{p^{\mathfrak{d}}}$. Given the sizes of the values involved, naive algorithms will work best, and hence the complexity of this algorithm will be

$$O(e \cdot \mathfrak{n}^2 \cdot \mathfrak{d}^2) = O(\mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.2.2 $Gao(\mathbf{c}, e)$

The method is called once and requires interpolating a polynomial using Lagrange interpolation, comput–ing the extended Euclidean algorithm and divisions in $\mathbb{F}_{p^{\mathfrak{d}}}$. When implemented naively, the overall cost of this algorithm, which is due to polynomial multiplications and divisions, will be

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

This complexity can be reduced by implementing polynomial arithmetic via fast Fourier transform (FFT), but it is likely that this only pays off for for large values of $\mathfrak{n}$, e.g. $\mathfrak{n} \gg 100$.

9.2.3 $ErrorCorrect(q, \mathbf{c}, e)$

This requires k iterations of the algorithm $BW(\mathbf{c}, e)/Gao(\mathbf{c}, e)$ and thus the complexity is either

$$O(k \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3) = O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3)$$

or

$$O(k \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2) = O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

We select using the Gao algorithm in what follows (and in our code).

9.2.4 $SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$

This requires a loop of $O(\mathfrak{n})$, where the main cost is multiplication and division of polynomials of degree at more $r = O(\mathfrak{n})$, with coefficients in $\mathbb{F}_{p^{\mathfrak{d}}}$. So a crude complexity estimate is given by

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.2.5 $Correct^{\mathbb{F}}(p, \mathbf{e})$

This is essentially the cost of $SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$ and so has complexity

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.2.6 $SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$

This requires k iterations of the algorithm $SynDecode^{\mathbb{F}}(p, S_{\mathbf{e}}(Z))$ and thus the complexity is

$$O(k \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3) = O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.2.7 $Correct^{GR}(q, \mathbf{e}$

This is essentially the cost of $SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$ and so has complexity

$$O(k \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3) = O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.2.8 $Share(a)$

This requires evaluating a polynomial of degree $\mathfrak{t}$ at $\mathfrak{n}$ distinct points in the Galois Ring, and thus naively has a complexity of

$$O(\mathfrak{d}^2 \cdot \mathfrak{t} \cdot \mathfrak{n}) = O(\mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.2.9 $OpenShare(\{\langle a \rangle_i\}_{i \in S})$

To create the polynomial $g_a(X)$ requires naively $O(\mathfrak{n} \cdot \mathfrak{t})$ multiplications in the Galois Ring, thus we have a complexity of

$$O(\mathfrak{d}^2 \cdot \mathfrak{t} \cdot \mathfrak{n}) = O(\mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.2.10 $Commit(m)$

This requires one hash function evaluation in order to produce an output of size $|Hash^{2 \cdot sec}| = 2 \cdot sec$. Thus we treat the complexity as unit time.

9.2.11 $Verify(c, o, m)$

This requires another hash function evaluation, and hence takes unit time.

9.2.12 $Solve(v)$

The execution of the sub–procedure $Solve_1$ requires $O(\mathfrak{d}^2)$ operations in the finite field $\mathbb{F}_{2^{\mathfrak{d}}}$. The execution of the main loop in the Hensel lifting requires $O(\log_2 k)$ operations in the Galois Ring. Hence, the overall complexity is

$$O(\log k \cdot \mathfrak{d}^2) = O(\log \log q \cdot \mathfrak{d}^2).$$

9.2.13 *Sqrt*(*v*, *p*)

The complexity of computing a square root is roughly $\mathfrak{d} \cdot \log_2 p$ multiplications in the Galois Ring $GR(p, F)$. Thus the overall complexity in terms of operations modulo $p < q$ is

$$O(\log p \cdot \mathfrak{d}^3).$$

9.2.14 *TreePRG.Gen*(*seed*, *d*)

This requires $O(2^d)$ executions of the PRG, in order to define seeds at each leaf and internal node of the tree. Thus the complexity is

$$O(2^d).$$

9.2.15 *TreePRG.GenSub*(*seed*, *lab*, *d*, *i*)

This subroutineis similar to *TreePRG.Gen* except it starts from depth i which is not necessarily the root. Thus the complexity is

$$O(2^{d-i}).$$

9.2.16 *TreePRG.Punc*(*seed*, *d*, *T*)

From Equation 30, we know the output of this subroutine has size $|D|$. Every output requires a prefix comparison of at most d bits for every element in T. Thus the complexity is

$$O\left(d \cdot |T|^2 \cdot \log_2\left(\frac{2^d}{|T|}\right)\right).$$

9.2.17 *TreePRG.PuncSub*(*D*, *lab*, *T*, *i*)

We use the same analysis as *TreePRG.Punc*, except for the set $T_i = T \setminus \{1, \ldots, 2^i\}$ and depth $d - i$. Thus the complexity is

$$O\left(d \cdot |T_i|^2 \cdot \log_2\left(\frac{2^{d-i}}{|T_i|}\right)\right).$$

9.2.18 *TreePRG.GenPunc*(*D*, *T*, *d*)

The subroutine *TreePRG.GenSub* needs to be executed $|D|$ times, thus the overall complexity is.

$$O(|D| \cdot O(2^d)).$$

But in practice the complexity is much better since the depth i is not always 0.

9.2.19 **Layer Zero Summary**

We summarize the above discussion in the following table:

Protocol	Rounds	Complexities Communication	Computation
BW(**e**, *e*)	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3)$
Gao(**e**, *e*)	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
ErrorCorrect(*q*, **e**, *e*)	–	–	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
SynDecode$^{\mathbb{F}}$(*p*, $S_{\mathbf{e}}(Z)$)	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3))$
Correct$^{\mathbb{F}}$(*p*, **r**)	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3))$

$SynDecode^{GR}(q, S_{\mathbf{e}}(Z))$	–	–	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3))$
$Correct^{GR}(q, \mathbf{r})$	–	–	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3))$
$Share(a)$	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$OpenShare(\{\langle a \rangle_i\}_{i \in S})$	–	–	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$Commit(m)$	–	–	$O(1)$
$Verify(c, o, m)$	–	–	$O(1)$
$Solve(v)$	–	–	$O(\log \log q \cdot \mathfrak{d}^2)$
$Sqrt(v, p)$	–	–	$O(\log p \cdot \mathfrak{d}^3)$
$TreePRG.Gen(seed, lab^*, d)$	–	–	$O(2^d)$
$TreePRG.GenSub(seed, lab, d, i)$	–	–	$O(2^{d-i})$
$TreePRG.Punc(seed, d, T)$	–	–	$O\left(d \cdot |T|^2 \cdot \log_2\left(\frac{2^d}{|T|}\right)\right)$
$TreePRG.PuncSub(D, lab, T, i)$	–	–	$O\left(d \cdot |T_i|^2 \cdot \log_2\left(\frac{2^{d-i}}{|T_i|}\right)\right)$
$TreePRG.GenPunc(D, T, d)$	–	–	$O(|D| \cdot O(2^d))$

Table 12: Complexity of Layer Zero Algorithms.

9.3 Layer One Algorithm and Protocol Complexities

9.3.1 $Synch\text{-}Broadcast(S, m)$

As discussed Section 7.2.1 in the text, this requires $\mathfrak{t} + 3$ rounds of communication. Note, as we are assuming this protocol is used within a synchronous protocol, even if all parties are honest we still require $\mathfrak{t} + 3$ rounds. It may appear that the parties can continue with the calling protocol after three rounds when all are honest, but each party does not know whether all the other parties are able to proceed. Thus the synchronicity of the parties in the calling protocol will be affected if some parties return after three rounds, and some need to wait.

The senders communication complexity is

$$O(\mathfrak{n} \cdot |m|).$$

The receivers communication complexity is $|m|$ data received from the sender, and the sending to $\mathfrak{n}$ players the *send* and *vote* messages (the first requiring $|m|$ bits, whilst the second $|Hash^{2 \cdot sec}| = 2 \cdot sec$ bits), and the receiving of at most $\mathfrak{n}$ *send* and *vote* messages. Thus the total amount of communication sent by a receiving player is $O(\mathfrak{n} \cdot |m| + \mathfrak{n} \cdot sec) = O(\mathfrak{n} \cdot |m|)$, resulting in a total communication complexity of

$$O(\mathfrak{n}^2 \cdot |m|).$$

The computational complexity is $O(1)$.

9.3.2 $RobustOpen(\mathcal{P}, \langle a \rangle^d)$

The round complexity of this protocol is equal to one. The communication complexity of this step is that each player needs to send a single ring element to player $\mathcal{P}$. Thus the total communication complexity is

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}).$$

The computational complexity of these protocols depend on whether one is executing the online error correction method, or one is waiting to receive enough shares to execute the *ErrorCorrect* in one go. In both cases the happy path computational complexity is (roughly) the same as that of *ErrorCorrect*$(q, \mathbf{c}, \mathfrak{t})$. The main advantage of using the second variant, for asynchronous networks, when $d + 2 \cdot \mathfrak{t} \leq \mathfrak{n}$ is that one can optimistically proceed with an execution of *ErrorCorrect*, whilst

waiting for slow players to respond. We model computational complexity on the happy path and hence have a computational complexity of

$$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.3.3 $RobustOpen(\mathcal{S}, \langle a \rangle^d)$

This has exactly the same round and computational complexity as the operation $RobustOpen(\mathcal{P}, \langle a \rangle^d)$, except now the communication complexity is increased by a factor of $|\mathcal{S}|$.

9.3.4 $BatchRobustOpen((\langle x_i \rangle^d)_{i=1}^{\ell})$

This has round complexity two, with the computational complexity dominated by the two executions of *ErrorCorrect* per player. In total each player sends $O(\mathfrak{n})$ ring elements in the protocol, making for a total communication complexity of

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2).$$

9.3.5 $VSS(\mathcal{P}_k, s, \mathfrak{t}, Corrupt)$

In [CCP22] the round complexity of the finite field variant of this scheme is indicated as four "rounds", which include three rounds of broadcast. The communication complexity is listed as $O(\mathfrak{n}^2)$ finite field elements of normal communication and $O(\mathfrak{n}^2)$ finite field elements needing to be broadcast.

Recall we are only giving complexities for the happy path of protocols. In this case we see that the protocol terminates after the second round of communication. In which case we see that the communication over normal channels (non–broadcast) channels in Round 1 is that the sender sends $2 \cdot (\mathfrak{t} + 1)$ values privately to each player. Then in the second round each player broadcasts $2 \cdot \mathfrak{n}$ values to all other players.

Thus on the happy path we have a round complexity of

$$\mathfrak{t} + 4,$$

a total communication complexity of

$$O(\mathfrak{n} \cdot \log q \cdot \mathfrak{d} + \mathfrak{n}^4 \cdot \log q \cdot \mathfrak{d}) = O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^4).$$

The computational complexity is bounded by each player needing to evaluate a polynomial of degree at most $\mathfrak{t}$ in $O(\mathfrak{n})$ points, i.e. roughly

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.3.6 $AgreeRandom(S, k)$

Each party in S needs to send one commitment, and then the opening value. Thus the round complexity is two, and the communication complexity is $O((k + sec) \cdot |S|)$, and the computational complexity is $O(1)$.

9.3.7 $AgreeRandom\text{-}w\text{-}Abort(\mathcal{S}, k)$

As well as executing $AgreeRandom(\mathcal{S}, k)$ this executes a second confirmation round of commitment and opening. Thus the total round complexity is four, and the communication complexity is again $O((k + sec) \cdot |\mathcal{S}|)$ and the computational complexity is $O(1)$.

9.3.8 $AgreeRandom\text{-}Robust(\mathcal{S}, k, \langle r \rangle)$

This is essentially (bar the hash function evaluation) just the same as $RobustOpen(\mathcal{S}, \langle r \rangle)$.

9.3.9 *PRSS.Init*()

This is the active–with–abort secure variant of *PRSS.Init*(). The algorithm in Figure 51 requires $\binom{\mathfrak{n}}{\mathfrak{t}}$ executions of *AgreeRandom-w-Abort*(A, sec). Assuming these are all done in parallel, this equates to four rounds, and a communication complexity of

$$O\left(\binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec \right).$$

The computational complexity depends also on the same exponential term, i.e.

$$O\left(\binom{\mathfrak{n}}{\mathfrak{t}} \right).$$

9.3.10 *PRSS.Init*(*Corrupt*)

The robust version *PRSS.Init*(*Corrupt*) is more involved. A total of $c \cdot \mathfrak{n}$ number of the *VSS* protocol are executed in parallel by each of the $\mathfrak{n}$ players, where

$$c = O\left(\frac{1}{\mathfrak{n}} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right),$$

i.e. we execute a total of $c \cdot \mathfrak{n}^2$ versions of *VSS* in parallel. After which the players extract a set of random secret shared values, by local computation. Finally a total of $\binom{\mathfrak{n}}{\mathfrak{t}}$ number of executions of *AgreeRandom-Robust* operations are executed; where again these last executions can be performed in parallel.

Thus we have a round complexity (on the happy path) of $(\mathfrak{t} + 4) + 1 = \mathfrak{t} + 5$. The total communication complexity is given by, as usually $\mathfrak{n} \ll \log q$ will hold in most instantiations,

$$O\left(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^6 \cdot \frac{1}{\mathfrak{n}} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} + \log q \cdot \mathfrak{d} \cdot \mathfrak{n} \cdot (\mathfrak{n} - \mathfrak{t}) \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right) = O\left(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right).$$

Finally the computational complexity is

$$O\left(\mathfrak{d}^2 \cdot \mathfrak{n}^4 \cdot \frac{1}{\mathfrak{n}} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} + \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right) = O\left(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right).$$

9.3.11 *PRSS.Next*()

This has zero round complexity (which is after all the point), but the computational complexity is

$$O\left(\log q \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right),$$

since each application of ψ requires $\log q$ AES evaluations, and the multiplication of the output into the stored values $f_A(\alpha_i)$ requires $O(\mathfrak{d}^2)$ operations, and in most instantiations we will have $\mathfrak{d}^2 \ll \log q$.

9.3.12 *PRSS.Check*(*cnt, Corrupt*)

Recall this is only ever called on an unhappy path, thus (as we are not focusing on the unhappy path complexity). However, for those interested: The main cost is the rounds needed to execute the synchronous broadcast. Assuming this is executed in parallel for all players, we have a total round complexity of $\mathfrak{t} + 4$. The message size, per player, of the broadcast is given by

$$O\left(\binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec \right).$$

Thus the total communication complexity is

$$O\left(\mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec\right).$$

The computational complexity is essentially $\mathfrak{n}$ times that of *PRSS.Next*().

9.3.13 *PRZS.Next*()

This is similar to *PRSS.Next*(), however now we execute per player $\mathfrak{t} \cdot \binom{\mathfrak{n}}{\mathfrak{t}}$ evaluations of the PRF χ, before multiplying the value into the stored values $\alpha_i^j \cdot f_A(\alpha_i)$. However, the multiplication of the $\mathbb{Z}/(q)$ values output by χ into the stored values requires $O(\mathfrak{d}^2)$ operations. Thus the computational complexity is given by, as $\mathfrak{d}^2 \ll \log q$,

$$O\left(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}}\right).$$

9.3.14 *PRSS.Check*($cnt, Corrupt$)

This is much like *PRSS.Check*($cnt, Corrupt$), however the message size per player in the broadcast is now of size

$$O\left(\mathfrak{t} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec\right).$$

Hence, the total communication complexity is

$$O\left(\mathfrak{n}^3 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec\right).$$

The computational complexity is essentially $\mathfrak{n}$ times that of *PRZS.Next*().

9.3.15 *PRSS-Mask.Next*($Bd, stat$)

The computational complexity, per player, is given by

$$O\left(\log q \cdot \binom{\mathfrak{n}}{\mathfrak{t}}\right).$$

9.3.16 *CoinFlip*($Corrupt$)

The *CoinFlip* operations requires the $\mathfrak{n}$ to execute one call to the *VSS* operation each, in parallel, followed by a call to *RobustOpen*($\{\mathcal{P}_1, \dots, \mathcal{P}_n\}, \langle x \rangle$). Thus the round complexity is $\mathfrak{t} + 5$. The total communication complexity is

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5 \ + \ \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2) = O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5).$$

The computational complexity per player is given by

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

9.3.17 Layer One Summary

We summarize the above discussion in the following table:

Protocol	Rounds	Complexities	
		Communication	Computation

Synch-Broadcast(S, m)	$\mathfrak{t} + 3$	$O(\mathfrak{n}^2 \cdot |m|)$	$O(1)$
RobustOpen($\mathcal{P}, \langle a \rangle^d$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
RobustOpen($\mathcal{S}, \langle a \rangle^d$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n} \cdot |\mathcal{S}|)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
BatchRobustOpen($(\langle x_i \rangle^d)_{i=1}^{\ell}$)	2	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
VSS($\mathcal{P}_k, s, \mathfrak{t}, Corrupt$)	$\mathfrak{t} + 4$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^4)$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
AgreeRandom($\mathcal{S}, k$)	2	$O((k + sec) \cdot |\mathcal{S}|)$	$O(1)$
AgreeRandom-w-Abort($\mathcal{S}, k$)	4	$O((k + sec) \cdot |\mathcal{S}|)$	$O(1)$
AgreeRandom-Robust($\mathcal{S}, k, \langle r \rangle$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n} \cdot |\mathcal{S}|)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
PRSS.Init()	4	$O(sec \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$	$O(\cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRSS.Init(Corrupt)	$\mathfrak{t} + 5$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5 \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRSS.Next()	–	–	$O(\log q \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRSS.Check(cnt, Corrupt)	$\mathfrak{t} + 3$	$O(\mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec)$	$O(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRZS.Next()	–	–	$O(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRZS.Check(cnt, Corrupt)	$\mathfrak{t} + 3$	$O(\mathfrak{n}^3 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot sec)$	$O(\log q \cdot \mathfrak{n}^2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
PRSS-Mask.Next(Bd, stat)	–	–	$O(\log q \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
CoinFlip(Corrupt)	$\mathfrak{t} + 5$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5)$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^3)$

Table 13: Complexity of Layer One Algorithms and Protocols.

9.4 Layer Two Algorithm and Protocol Complexities

9.4.1 *MPC^S.Init*()

This is identical to the cost of *PRSS.Init*() or *PRSS.Init*(*Corrupt*) depending on which one is selected.

9.4.2 *MPC^S.GenTriples*(*Dispute*)

On the happy path this protocol consists of three *PRSS* evaluations, one *PRZS* evaluation, a parallel execution of *Synch-Broadcast* for a message of size $\log q \cdot \mathfrak{d}$, and the application of the algorithm *ErrorCorrect*.

The communication complexity (on the happy path) is thus

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3),$$

and the computational complexity is

$$O\left(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} + \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2\right) = O\left(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}}\right).$$

The the round complexity is $\mathfrak{t} + 3$, however, in practice one will execute many of the broadcast calls in parallel and thus the overall round complexity per triple is close to zero.

9.4.3 *MPC^S.NextRandom*(*Dispute*)

This is simply an execution of *PRSS.Next*(), and so has the same complexity.

9.4.4 *ShareDispute*($\mathcal{P}_i, s, d, Dispute$)

This requires evaluation of a degree d polynomial at $\mathfrak{n}$ points over $GR(q, F)$ and so requires a computa–tional cost of

$$O(d \cdot \mathfrak{d}^2 \cdot \mathfrak{n})$$

from the sending player. The communications requires one round, and requires the sending of one ring element per player, i.e.

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}).$$

9.4.5 *LocalSingleShare*$(\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute)$

We first examine a single (non parallel) execution of this algorithm as it is described in Figure 60 (the execution of the parallel version we discuss afterwards). We examine each step requiring computation or communication in turn:

- Line 2: This requires ℓ parallel executions of *ShareDispute*$(\mathcal{P}_i, s, \mathfrak{t}, Dispute)$. Thus it requires one round of communication, a communication complexity of $O(\ell \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n})$ and a computational complexity of $O(\ell \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$.
- Line 3: This can be executed in parallel with the previous step, and so costs no extra rounds. The communication and computational complexity is given by $O(m \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n})$ and $O(m \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$.
- Line 4: This is simply an execution of the *CoinFlip* protocol. As determined above this requires $\mathfrak{t} + 5$ rounds, and costs a communication complexity of $O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5)$ and a computational complexity of $O(\mathfrak{d}^2 \cdot \mathfrak{n}^3)$.
- Line 6: This requires the parallel execution of m evaluations of the following steps:
  - A maximum of $\mathfrak{n}$ send data to all the other players using *Synch-Broadcast*. These can all be done in parallel, with each message being of size $O(\ell \cdot \log q \cdot \mathfrak{d})$. Thus these steps will consume $\mathfrak{t} + 3$ rounds, and consume bandwidth of $O(m \cdot \ell \cdot \log q \cdot \mathfrak{n}^2)$. The computation requires in this step is roughly $O(m \cdot \ell \cdot \mathfrak{d}^2)$.

We make the reasonable assumption that $m \ll \ell$, since the point of this protocol is to agree on many sharings in one go (i.e. ℓ will be very large in comparison to m). Hence the complexity of line 3 can be ignored. We also assume that $\mathfrak{n}^4 \ll \ell$ as well, for much the same reasons. We can also assume that $m \ll \mathfrak{n}^3$, since this protocol will only be applied when $\mathfrak{n} > 10$ and we know $m \leq dist = 80$.

> **Parameter Choice 6: Size of ℓ in *LocalSingleShare***
>
> We assume $\ell > \mathfrak{n}^4$ This is mainly for our complexity estimates to hold, but also this makes sense from an implementations stand point as well.

Summing up the above analysis: We find a round complexity of

$$1 + (\mathfrak{t} + 5) + (\mathfrak{t} + 3) = 2 \cdot \mathfrak{t} + 9.$$

A communication complexity of

$$O\left(\ell \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n} + \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^5 + m \cdot \ell \cdot \log q \cdot \mathfrak{n}^2\right) = O(\ell \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n}).$$

The computational complexity is given by

$$O\left(\ell \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 + \mathfrak{d}^2 \cdot \mathfrak{n}^3 + m \cdot \ell \cdot \mathfrak{d}^2\right) = O(\ell \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

When we execute *LocalSingleShare* in parallel, with each player $\mathcal{P}_i$ executing it to enter ℓ values into the system, the round complexity stays the same, and the communication and computational complexities simply increase by a factor of $\mathfrak{n}$.

Note, that in our code we do not actually execute lines 2, 3 and 6 in parallel, thus in practice we have one extra round. As this round is eventually amortized away, this makes no significant difference to the final protocol round complexity.

9.4.6 *LocalDoubleShare*($\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute$)

This is essentially the same as *LocalSingleShare*, except one processes two sharings at once and performs some additional checks. All these extra factors are swallowed in the constant of the big–Oh notation.

9.4.7 *SingleSharing.Init*(*Dispute*)/*DoubleSharing.Init*(*Dispute*)

We do not count the execution of the initialization routines into the complexity counts, as we shall instead amortize the call's here into the *Next* operations discussed next.

9.4.8 *SingleSharing.Next*(*Dispute*)/*DoubleSharing.Next*(*Dispute*)

These operations take a batch of ℓ values per player, obtained by executing *LocalSingleShare* (resp. *LocalDoubleShare*) in parallel $\mathfrak{n}$ times. Then these $\ell \cdot \mathfrak{n}$ values are used to produce $\ell \cdot (\mathfrak{n} - \mathfrak{t}) = O(\ell \cdot \mathfrak{n})$ output values. Thus the complexity per invocation of the respective *Next*() command is actually the complexity of the parallel invocation of the *LocalSingleShare* (resp. *LocalDoubleShare*) commands, divided by $\ell \cdot \mathfrak{n}$, i.e. the cost of a single invocation of *LocalSingleShare* (resp. *LocalDoubleShare*) divided by ℓ. The round complexity gets amortized into an $O(1)$ term, which in practice is on average approximately zero (as ℓ is large).

9.4.9 *MPC^L.Init*(*Dispute*)

As above we consider this call to be amortized into the following calls to *SingleSharing.Next*(*Dispute*) and/or *DoubleSharing.Next*(*Dispute*).

9.4.10 *MPC^L.GenTriples*(*Dispute*)

To generate each triple requires a total of two calls to the procedure *SingleSharing.Next*(*Dispute*) and one call to the procedure *DoubleSharing.Next*(*Dispute*). Then each parties sends one sharing value to all other parties, and finally *ErrorCorrect* is called. The communication complexity is $O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$, and the computational complexity is $O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$. The round complexity is approximately one, however, the round complexity can be amortized over many calls, and thus we can ignore it if many executions are done in parallel.

9.4.11 *MPC^L.NextRandom*(*Dispute*)

This is simply a renaming of *SingleSharing.Next*(*Dispute*), and thus has the same complexity.

9.4.12 **Layer Two Summary**

We summarize the above discussion in the following table:

		Complexities	
Protocol	Rounds	Communication	Computation
MPC^S.Init()	Same as *PRSS.Init*() or *PRSS.Init*(*Corrupt*)		
MPC^S.GenTriples(*Dispute*)	$\mathfrak{t} + 3/\approx 0$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
MPC^S.NextRandom(*Dispute*)	–	–	$O(\log q \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
ShareDispute($\mathcal{P}_i, s, d, Dispute$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(d \cdot \mathfrak{d}^2 \cdot \mathfrak{n})$
LocalSingleShare($\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute$)	$2 \cdot \mathfrak{t} + 9$	$O(\ell \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\ell \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
LocalDoubleShare($\mathcal{P}_i, (s_1, \ldots, s_\ell), Dispute$)	$2 \cdot \mathfrak{t} + 9$	$O(\ell \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\ell \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$

SingleSharing.Init(*Dispute*)	–	–	–
SingleSharing.Next(*Dispute*)	$(2 \cdot \mathfrak{t} + 9)/\ell \approx 0$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
DoubleSharing.Init(*Dispute*)	–	–	–
DoubleSharing.Next(*Dispute*)	$(2 \cdot \mathfrak{t} + 9)/\ell \approx 0$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
MPC^L.*Init*(*Dispute*)	–	–	–
MPC^L.*GenTriples*(*Dispute*)	$\approx 1/\approx 0$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$
MPC^L.*NextRandom*(*Dispute*)	≈ 0	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n})$	$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2)$

Table 14: Complexity of Layer Two Algorithms and Protocols.

9.5 Layer Three Algorithm and Protocol Complexities

9.5.1 *MPC.Open*($\langle x \rangle$)

This is exactly the same as *RobustOpen*, for a receiving set of size *n*, hence the round complexity is one, the communication complexity is

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$$

and the computational complexity is

$$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.5.2 *MPC.Mult*($\langle x \rangle, \langle y \rangle$)

This requires one call to *MPC.GenTriples* and two parallel calls to *MPC.Open* thus the round complexity is effectively equal to one. The communication and computational complexity depends on which regime (*nSmall* vs *nLarge*) one is in, and whether one considers the online and offline phases to be distinct or not. Again we only consider the complexity on the more important happy path. The online complexity is the same irrespective of which regime one is in, and it is asymptotically the same as for *MPC.Open*.

The combined offline and online complexity in the *nSmall* regime are for communication

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3 + \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2) = O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3),$$

and for computation

$$O\left(\log q \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} + \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \right) = O\left(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \right).$$

In the *nLarge* regime the combined offline and online communication and computational complexities are given by

$$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n} + \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2) = O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$$

and

$$O(\mathfrak{d}^2 \cdot \mathfrak{n}^2 + \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2) = O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2).$$

9.5.3 *MPC.GenBits*(v)

Recall this only works in the *nSmall* regime when $q = p_1 \cdots p_k$. Also recall, there are different algorithms depending on whether $q = p^k$ or $q = p_1 \cdots p_k$. Per bit generated the routine requires one call to *MPC.NextRandom*, one call to *MPC.Mult*, and depending on the value of q either one (q a power of two) or two (odd q) calls to *MPC.Open*, plus some book keeping. The rounds can be merged for each call to *MPC.GenBits*, thus irrespective of v, the round complexity is two or three. The communication and computational complexities are dominated by the calls to *MPC.Mult*.

9.5.4 Layer Three Summary

We summarize the above discussion in the following table, for *MPC.Mult* and *MPC.GenBits* we present the combined online and offline complexities:

		Complexities			
		nSmall		nLarge	
Protocol	Rounds	Communication	Computation	Communication	Computation
MPC.Open($\mathcal{U}, \langle x \rangle$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
MPC.Mult($\langle x \rangle, \langle y \rangle$)	1	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$	$O(\log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
MPC.GenBits(v)	2 or 3	$O(v \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(v \cdot \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$	$O(v \cdot \log q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(v \cdot \log q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$

Table 15: Complexity of Layer Three Algorithms and Protocols.

Note the *nSmall* communication complexity is larger than that of *nLarge*, which seems counter intuitive given that *nSmall* uses PRSS to reduce communication. However this can be explained by two facts:

- The *nSmall* profile achieves $\mathfrak{t} < \mathfrak{n}/3$ as opposed to the $\mathfrak{t} < \mathfrak{n}/4$ of the *nLarge* profile. Thus "more" is being achieved in the threshold profile *nSmall*.
- The low communication costs in *nLarge* is caused by the amortization over the $\ell = O(\mathfrak{n}^4)$ parallel repetitions. If $\mathfrak{n}$ is very large, then achieving this level of amortization may not be feasible, in which case the communication complexity will increase by however much one decreases ℓ.

9.6 Layer Four Algorithm and Protocol Complexities

Recall here we utilize the modulus Q (BGV, BFV and TFHE when we use the second threshold decryption algorithm) or $\overline{Q}$ (for TFHE and the first threshold decryption algorithm).

9.6.1 *MPC.NewHope*(N, B)

The cost of this is the cost of the call to *MPC.GenBits*($2 \cdot N \cdot B$). The rest of the algorithm is local computation, which is marginal, and hence does not affect the overall computational complexity. We only ever use this algorithm for BGV and BFV, for threshold profile *nSmall*. The round complexity of *MPC.GenBits* is three in this case.

9.6.2 *MPC.TUniform*($N, -2^b, 2^b$)

Just like for *MPC.NewHope*($\cdot, \cdot$), the cost of this operation is the cost of the call to *MPC.GenBits*($N \cdot (b+1)$). In this case the round complexity of *GenBits* is two. The rest of the algorithm is local computation, which is marginal, and hence does not affect the overall computational complexity.

9.6.3 *BGV.Threshold-KeyGen*(...) and *BFV.Threshold-KeyGen*(...)

Due to our way of generating random bits when q is a produce of odd primes in our MPC engine; this algorithm can only be called in threshold profile *nSmall*. The calls to *MPC.NewHope* can all be performed in parallel (resulting in a total round complexity of three for these calls). The rounds for opening $\mathfrak{pf}_a, \mathfrak{pf}'_a$ and those for the multiplication, can also all be done together. As can the rounds to open $\mathfrak{pf}_b$ and $\mathfrak{pf}'_b$. Thus the total number of rounds is five.

The communication complexity of *BGV.Threshold-KeyGen* is given by

$$O(N \cdot (2 \cdot B + 1) \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3) + O(N \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$$
$$= O(N \cdot B \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3).$$

The computational complexity is given by

$$O(N \cdot (2 \cdot B + 1) \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}}) + O(N \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$$

$$= O(N \cdot B \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}}).$$

9.6.4 *BGV.Threshold-Dec*($\mathfrak{ct}$, $\langle\mathfrak{sf}\rangle$) **and** *BFV.Threshold-Dec*($\mathfrak{ct}$, $\langle\mathfrak{sf}\rangle$)

The dominating cost here in terms of complexity, and indeed the only part which requires interaction is the *RobustOpen* execution; which is evaluated N times in parallel. Here we assume that the execution of the N openings is performed in one round, and not via the two round *BatchRobustOpen* protocol which batches openings together. Assuming all $\mathfrak{n}$ parties obtain the output, this operation takes one round of communication, has a communication complexity of

$$O(\log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2 \cdot N)$$

and a computational complexity of

$$O(\log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot N).$$

9.6.5 *TFHE.Threshold-KeyGen*($\dots$)

Assuming all the secret shared random bits are produced in one batch, which requires two rounds, the rest of the protocol requires three additional rounds; one to open *seed*, one to open (in parallel) $\mathfrak{pf}_b$, $PKSK_i$, $KSK_{i,j}$, BK_i and $\overline{BK}_i$, and one to perform the multiplications needed in the execution of the *MPC.Enc*^{GGSW} sub–routine (which we assume are done in parallel). Thus a total of five rounds in total are required.

The communication and computational complexity are dominated by the need to generate triples, in order to produce the requisite number of bits. The number of triples required is (roughly)

$$O(\ell \cdot \overline{w} \cdot \overline{N}^2).$$

See below for a more precise analysis. This means the communication and computational complexities are given by

$$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$$

and

$$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$$

in the *nSmall* regime, and

$$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$$

and

$$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$$

in the *nLarge* regime. The $(\overline{Q}, \overline{w}, \overline{N})$ terms can be replaced by (Q, w, N) if we are using the second threshold decryption algorithm here.

9.6.6 *TFHE.Threshold-Dec-1*($\mathfrak{ct}$, PK, $\langle\mathfrak{sf}\rangle$)

This algorithms complexity is dominated by the application of *TFHE.SwitchSquash* and then *MPC.Open*. Thus, the round complexity is one, the computational complexity is

$$O(\ell \cdot \overline{w}^2 \cdot \overline{v_{bk}} \cdot \overline{N} \cdot \log \overline{N}),$$

whilst the communication complexity is

$$O(\log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2).$$

9.6.7   *XOR*($\langle a \rangle$, $\langle b \rangle$)

This has the same complexity as *MPC.Mult*.

9.6.8   *BitAdd*$(((\langle a_i \rangle)_{i=0}^{K-1}, ((\langle b_i \rangle)_{i=0}^{K-1})$

We call this algorithm when $Q = 2^K$, and so the number of rounds is $2 \cdot \log \log Q + 1$, in which one executes $O(\log Q \cdot \log \log Q)$ multiplication operations.

9.6.9   *BitSum*$(((\langle a_i \rangle)_{i=0}^{k-1})$

This is a purely local operation, and so is essentially for free.

9.6.10   *BitDec*($\langle a \rangle$)

This requires a call to *MPC.GenBits* (consuming two rounds) to generate K secret shared random bits, one call to *MPC.Open* (consuming one round) and then a call to *BitAdd*. Thus we require $2 \cdot \log \log Q + 4$ rounds of communication. The communication and computational complexity is dominated by the call to *BitAdd*.

9.6.11   *TFHE.Threshold-Dec-2*($\mathfrak{ct}$, $\langle \mathfrak{sk} \rangle$)

Apart from the *BitDec* operation this requires two extra rounds; one to implemen the secure multiplication and one to execute *MPC.Open*.

9.6.12   *ReShare*($S_1, S_2, \langle \mathfrak{sk} \rangle^{S_1}$)

The round complexity here is (assuming $S_1 \neq S_2$) three plus the round complexity of two synchronous broadcasts, and is thus

$$3 + 2 \cdot (\mathfrak{t} + 3) = 2 \cdot \mathfrak{t} + 9.$$

The main computational complexity comes from the application of *SynDecode*^{GR} and is thus

$$O(\log q \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3).$$

The communication complexity is, assuming parallel execution of the various *Synch-Broadcast*s, is around

$$O(\mathfrak{n}^2 \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d} \cdot \log Q),$$

where $\dim_{\mathfrak{sk}}$ is the dimension of the secret key. This is N in the case of BGV/BFV. For TFHE it is ℓ if $\overline{flag}$ is unset, or $\ell + \overline{w} \cdot \overline{N}$ if $\overline{flag}$ is set.

9.6.13   Layer Four Summary

We summarize the above discussion in the following table:

| | | Complexities | | |
| | | | nSmall/ nLarge | |
Protocol	Rounds	Communication		Computation

226

		nSmall / nLarge	
$MPC.NewHope(N, B)$	3	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	–	–	–
$MPC.TUniform(N, -2^b, 2^b)$	2	$O(N \cdot b \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(N \cdot b \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	2	$O(N \cdot b \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(N \cdot b \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$BGV.Threshold\text{-}KeyGen(\ldots)$	5	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	–	–	–
$BFV.Threshold\text{-}KeyGen(\ldots)$	5	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(N \cdot B \cdot \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	–	–	–
$BGV.Threshold\text{-}Dec(\mathfrak{ct}, \langle \mathfrak{sk} \rangle)$	2	$O(\log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2 \cdot N)$	$O(\log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot N)$
	–	–	–
$BFV.Threshold\text{-}Dec(\mathfrak{ct}, \langle \mathfrak{sk} \rangle)$	2	$O(\log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2 \cdot N)$	$O(\log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2 \cdot N)$
	–	–	–
$TFHE.Threshold\text{-}KeyGen(\ldots)$	5	$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	5	$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\ell \cdot \overline{w} \cdot \overline{N}^2 \cdot \log \overline{Q} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$TFHE.Threshold\text{-}Dec\text{-}1(\mathfrak{ct}, PK, \langle \mathfrak{sk} \rangle)$	1	$O(\log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\ell \cdot \overline{w}^2 \cdot \overline{\nu_{bk}} \cdot \overline{N} \cdot \log \overline{N})$
	1	$O(\log \overline{Q} \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\ell \cdot \overline{w}^2 \cdot \overline{\nu_{bk}} \cdot \overline{N} \cdot \log \overline{N})$
$XOR(\langle a \rangle, \langle b \rangle)$	1	$O(\log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O(\log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	1	$O(\log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O(\log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$BitAdd((\langle a_i \rangle)_{i=0}^{K-1}, (\langle b_i \rangle)_{i=0}^{K-1})$	$2 \cdot \log \log Q + 1$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	$2 \cdot \log \log Q + 1$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$BitSum((\langle a_i \rangle)_{i=0}^{K-1})$	–	–	–
	–	–	–
$BitDec(\langle a \rangle)$	$2 \cdot \log \log Q + 4$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	$2 \cdot \log \log Q + 4$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$TFHE.Threshold\text{-}Dec\text{-}2(\mathfrak{ct}, \langle \mathfrak{sk} \rangle)$	$2 \cdot \log \log Q + 6$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^3)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n} \cdot \binom{\mathfrak{n}}{\mathfrak{t}})$
	$2 \cdot \log \log Q + 6$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d} \cdot \mathfrak{n}^2)$	$O((\log Q)^2 \cdot \log \log Q \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^2)$
$ReShare(S_1, S_2, \langle \mathfrak{sk} \rangle^{S_1})$	$2 \cdot \mathfrak{t} + 9$	$O(\mathfrak{n}^2 \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d} \cdot \log Q)$	$O(\log q \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3)$
	$2 \cdot \mathfrak{t} + 9$	$O(\mathfrak{n}^2 \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d} \cdot \log Q)$	$O(\log q \cdot \dim_{\mathfrak{sk}} \cdot \mathfrak{d}^2 \cdot \mathfrak{n}^3)$

Table 16: Complexity of Layer Four Algorithms and Protocols. The top complexities for each row are for the profiles *nSmall*, whereas the bottom of each row are for the profiles *nLarge*.

9.6.14 Concrete Complexity of Key Generation

The above analytic complexities for key generation can seem a little opaque, and in practice one executes key generation for concrete FHE parameter sets, and thus asymptotics may not give the correct feeling of the complexity. Thus it is perhaps instructive to examine the concrete complexity of the key generation methods for our particular parameter sets; which are by–far the most complex and time consuming of our threshold operations.

The key concrete measure of performance in key generation is the number of multiplication triples which need to be generated. Thus we concentrate solely on this metric. We write $|NewHope(N, B)|_T$ etc to express the number of triples needed to execute the algorithm $NewHope(N, B)$. This allows us to recursively build up the formulae for $|BGV.Threshold\text{-}KeyGen()|_T$ using the following values:

$$|MPC.Mult(\cdot, \cdot)|_T = 1,$$
$$|MPC.GenBits(v)|_T = v \cdot |MPC.Mult(\cdot, \cdot)|_T \quad \text{(plus } \epsilon \text{ for BGV and BFV),}$$
$$|MPC.NewHope(N, B)|_T = |MPC.GenBits(2 \cdot N \cdot B)|_T,$$

$$|BGV.Threshold\text{-}KeyGen(N, Q, P, B, R)|_T = |MPC.NewHope(N, 1)|_T$$
$$+ 2 \cdot |MPC.NewHope(N, B)|_T$$
$$+ N \cdot |MPC.Mult(\cdot, \cdot)|_T,$$
$$|BFV.Threshold\text{-}KeyGen(N, Q, P, B, R)|_T = |BGV.Threshold\text{-}KeyGen(N, Q, P, B, R)|_T,$$

$$|MPC.TUniform(N, -2^b, 2^b)|_T = |MPC.GenBits(N \cdot (b + 2))|_T,$$
$$|MPC.Enc^{LWE}(\cdots, \ell, \cdot)|_T = |MPC.TUniform(1, -2^{b_\ell}, 2^{b_\ell})|_T,$$

227

$$|MPC.Enc^{GLWE}(\cdots, N, w, \cdot)|_T = |MPC.TUniform(N, -2^{b_{w \cdot N}}, 2^{b_{w \cdot N}})|_T,$$
$$|MPC.Enc^{Lev}(\cdots, \ell, \nu, \cdot)|_T = \nu \cdot |MPC.Enc^{LWE}(\cdots, \ell, \cdot)|_T,$$
$$|MPC.Enc^{GLev}(\cdots, N, w, \nu, \cdot)|_T = \nu \cdot |MPC.Enc^{GLWE}(\cdots, N, w, \cdot)|_T,$$
$$|MPC.Enc^{GGSW}(\cdots, N, w, \cdot, \nu, \cdot)|_T = w \cdot N \cdot |MPC.Mult(\cdot, \cdot)|_T$$
$$+ (w + 1) \cdot |MPC.Enc^{GLev}(\cdots, N, w, \nu, \cdot)|_T,$$

$$|TFHE.Threshold\text{-}KeyGen(\cdots, \hat{\ell}, \ell, N, w, \cdots,$$
$$\cdots, \nu_{pksk}, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \cdot, \overline{\nu_{bk}}, \cdot, \overline{flag})|_T = |MPC.GenBits(\ell)|_T$$
$$+ |MPC.GenBits(\hat{\ell})|_T$$
$$+ |MPC.TUniform(\hat{\ell}, -2^{b_{\hat{\ell}}}, 2^{b_{\hat{\ell}}})|_T$$
$$+ \hat{\ell} \cdot |MPC.Enc^{Lev}(\cdots, \ell, \nu_{pksk})|_T$$
$$+ w \cdot |MPC.GenBits(N)|_T$$
$$+ w \cdot N \cdot |MPC.Enc^{Lev}(\cdots, \ell, \nu_{ksk})|_T$$
$$+ \overline{flag} \cdot \overline{w} \cdot |MPC.GenBits(\overline{N})|_T$$
$$+ \ell \cdot |MPC.Enc^{GGSW}(\cdots, N, w, \cdot, \nu_{bk}, \cdot)|_T$$
$$+ \ell \cdot \overline{flag} \cdot |MPC.Enc^{GGSW}(\cdots, \overline{N}, \overline{w}, \cdot, \overline{\nu_{bk}}, \cdot)|_T,$$

where we have interpreted $\overline{flag} = false$ to be the integer zero, and $\overline{flag} = true$ to be the integer one.

We can then plug in our parameters for BGV, BFV and TFHE, given in tables Table 5 and Table 7, to find the exact value of the number of triples needed to be generated, in order to perform threshold key generation for the threshold parameters of our three FHE schemes. These values are given in Table 17.

Scheme	Table	Number Triples
BGV/BFV	Table 5	458,752
TFHE, $type = LWE, P = 8, \overline{flag} = false$	Table 7	53,432,222
TFHE, $type = LWE, P = 8, \overline{flag} = true$	Table 7	469,706,654
TFHE, $type = LWE, P = 32, \overline{flag} = false$	Table 7	21,867,500
TFHE, $type = LWE, P = 32, \overline{flag} = true$	Table 7	741,538,796
TFHE, $type = F\text{-}GLWE, P = 8, \overline{flag} = false$	Table 7	48,891,728
TFHE, $type = F\text{-}GLWE, P = 8, \overline{flag} = true$	Table 7	430,102,352
TFHE, $type = F\text{-}GLWE, P = 32, \overline{flag} = false$	Table 7	19,577,758
TFHE, $type = F\text{-}GLWE, P = 32, \overline{flag} = true$	Table 7	683,338,654

Table 17: Number of Triples Required for Threshold Key Generation.

As one can see the number are very large in terms of the number of triples which need to be computed. In Appendix B we present a different methodology, in the threshold regime $nSmall$, to perform threshold key generation. The idea behind the modified method, is to replace the calls to $MPC.NewHope$ and $MPC.TUniform$ with a call to $PRSS\text{-}Mask.Next$ in order to produce the underlying LWE noise samples. This however entails changing the underlying FHE parameters, and so incurs a computational cost during homomorphic operations.

The main issue with the number of triples is one of storage, especially if the offline phase is executed before the online phase. Thus another solution, in order to avoid storing a large number of triples, one could, within the offline phase, preprocess the shared random bits, and shares from the distributions $TUniform$ and $NewHope$ and just store these. This signficantly reduces the storage from billions of secret sharings (recall each triple is itself three sharings) down to at most tens of millions. To see the effect of this we present in Table 18 a table giving the storage required of the different offline data for

the different parameter sets. Note, the (potentially) billions of triples still need to be generated, they just do not need to be stored with this modification. It is this modification to the protocols which we adopt in our implementation[22].

Scheme	Bits	Number		
		Triples	NewHope	TUniform
BGV/BFV	0	65536	3	0
TFHE, $type = LWE, P = 8, \overline{flag} = false$	3,998	1,896,448	0	2,851,840
TFHE, $type = LWE, P = 8, \overline{flag} = true$	8,094	5,689,344	0	17,075,200
TFHE, $type = LWE, P = 32, \overline{flag} = false$	7,148	4,112,384	0	8,255,488
TFHE, $type = LWE, P = 32, \overline{flag} = true$	11,244	8,224,768	0	32,929,792
TFHE, $type = F\text{-}GLWE, P = 8, \overline{flag} = false$	3,920	1,736,704	0	2,611,200
TFHE, $type = F\text{-}GLWE, P = 8, \overline{flag} = true$	8,016	5,210,112	0	15,636,480
TFHE, $type = F\text{-}GLWE, P = 32, \overline{flag} = false$	7,070	3,792,896	0	7,602,176
TFHE, $type = F\text{-}GLWE, P = 32, \overline{flag} = true$	11,166	7,585,792	0	30,359,552

Table 18: Number of Bits, Triples and Disitribution Samples Required for Threshold Key Generation When Using Less Offline Storage.

9.7 Layer Five Algorithm and Protocol Complexities

We also here elaborate on the proof sizes for the different proof

9.7.1 *CRS-Gen.Init*($sec, \bar{q}, \bar{d}, \bar{B}$)

There is nothing to analyze here.

9.7.2 *CRS-Gen.Update*(pp_{j-1})

This is a protocol between a contributor, and the other parties. There is a single round of communication from the contributor to the other parties, this consists of

$$(2 \cdot \bar{B}_1) \cdot |\mathbb{G}| + (\bar{B}_1) \cdot |\hat{\mathbb{G}}|,$$

where $|\mathbb{G}|$ denote the size of an element in the first pairing source group, and $|\hat{\mathbb{G}}|$ denotes the size of an element in the second pairing source group. Thus, assuming an elliptic curve $\mathbb{G} = E(\mathbb{F}_t)$ and extension degree k for $\hat{\mathbb{G}} = E(\mathbb{F}_{t^k})$ and point compression is used, these require $O(\log t)$ and $O(k \cdot \log t)$ bits to represent them. Thus the communication complexity (per verifying party) is given by

$$O(k \cdot \bar{B}_1 \cdot \log_2 t).$$

The computational complexity of each verifier is dominated by $2 \cdot \bar{B}_1$ point multiplications in $\mathbb{G}$, $\bar{B}_1$ point multiplications in $\hat{\mathbb{G}}$, and two pairing product equations in order to check the verification equations (35). The contributor needs to compute $2 \cdot \bar{B}_1$ point multiplications in $\mathbb{G}$ and $\bar{B}_1$ point multiplications in $\hat{\mathbb{G}}$ to compute pp_j from pp_{j-1}.

9.7.3 *CRS-Gen.Output*()

There is nothing to analyze here.

---

[22]The changes to the protocol descriptions are immediate, but we opted to present the protocols in this document in there most easy to understand manner; i.e. without this storage optimization.

9.7.4 *VC-Prove-1*$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$

The prover is required to compute $\bar{B}_1$ point multiplications in $\mathbb{G}$ and $\bar{B}_1$ point additions in $\hat{\mathbb{G}}$. The point multiplications will dominate, and so we approximate this by

$$\bar{B}_1 \cdot \mathbb{G}.$$

The above is a loose bound since the prover actually compute a multi–exponentiations with $\bar{B}$ generators in a single multi–exponentiation.

9.7.5 *VC-Verify-1*$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), prf)$

The verifier is required to compute $\bar{B}_1$ point multiplications in both $\mathbb{G}$ and $\hat{\mathbb{G}}$, as well as a single pairing product equation of three terms. We express this as

$$\bar{B}_1 \cdot (\mathbb{G} + \hat{\mathbb{G}}) + \mathbf{PP}.$$

The proof size of this proof system is given by (assuming an elliptic curve $\mathbb{G} = E(\mathbb{F}_t)$ and extension degree k for $\hat{\mathbb{G}} = E(\mathbb{F}_{t^k})$ and point compression is used.)

$$sz = 2 \cdot |\mathbb{G}| + |\hat{\mathbb{G}}| = (2 + k) \cdot \log_2 t.$$

9.7.6 *VC-Prove-2*$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$

The prover is required to compute $6 \cdot \bar{B}_1 + 2 \cdot N$ point multiplications in $\mathbb{G}$ and $\bar{B}_1 + 3 \cdot N$ point multiplications in $\hat{\mathbb{G}}$. We denote this by

$$(6 \cdot \bar{B}_1 + 2 \cdot N) \cdot \mathbb{G} + (\bar{B}_1 + 3 \cdot N) \cdot \hat{\mathbb{G}}.$$

The above is a loose bound since the prover actually computes several multi–exponentiations with $\bar{B} + 1$ generators in each multi–exponentiation.

9.7.7 *VC-Verify-2*$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), prf)$

The verifier is required to compute 128 exponentiations in $\hat{\mathbb{G}}$ and a product of eight pairings. We denote this by

$$128 \cdot \hat{\mathbb{G}} + 8 \cdot \mathbf{PP}.$$

The proof size of this proof system is given by (again, assuming an elliptic curve $\mathbb{G} = E(\mathbb{F}_t)$ and extension degree k for $\hat{\mathbb{G}} = E(\mathbb{F}_{t^k})$ and point compression is used.)

$$sz = 8 \cdot |\mathbb{G}| + 5 \cdot |\hat{\mathbb{G}}| = (8 + 5 \cdot k) \cdot \log_2 t.$$

9.7.8 *MPCitHead-Prove-1*$((A, \mathbf{s}), \mathbf{b})$

Proof generation has time complexity dominated by

$$O(d \cdot B \cdot \tau \cdot (n - 1)).$$

The size of the proof as we have written is (in the absolutely worst case, where all bar one transcript is rejected):

$$sz \leq |\mathcal{H}_2| + |\mathcal{H}_4| + \tau \cdot sec \cdot \log_2 \left(\frac{2^{\lceil \log_2 M \rceil}}{\tau} \right)$$

$$+ \eta \cdot (1 + |\mathcal{H}_1| + |\mathcal{H}_3|)$$

$$+ \tau \cdot sec \cdot \left(n - 1 + \log_2 \left(2^{\lceil \log_2 n \rceil}\right)\right)$$
$$+ \tau \cdot (B \cdot \log_2 A + B + |Commit|).$$

Note, as we use a hash based commitment we have

$$|\mathcal{H}_1| = |\mathcal{H}_2| = |\mathcal{H}_3| = |\mathcal{H}_4| = |Commit| = 2 \cdot sec.$$

We also have $A = 2^a$ so this formula becomes:

$$sz \leq 4 \cdot sec \cdot (1 + \eta) + \eta + \tau \cdot B \cdot (a + 1)$$
$$+ sec \cdot \tau \cdot \left(2 + \log_2 \left(\frac{2^{\lceil \log_2 M \rceil}}{\tau}\right) + \left(n - 1 + \log_2 \left(2^{\lceil \log_2 n \rceil}\right)\right)\right). \tag{50}$$

To obtain specific parameters one strategy would be to minimize the proof time estimate above, subject to a specific desired upper bound on the proof size.

9.7.9 *MPCitHead-Verify-1*$((A, \mathbf{s}), prf)$

The verification takes roughly the same time as the proof generation, thus the complexity is

$$O(d \cdot B \cdot \tau \cdot (n - 1)).$$

9.7.10 *MPCitHead-Prove-2*$((A, \mathbf{s}), \mathbf{b})$

Proof generation has time complexity dominated by

$$O\left(B \cdot \left(N^D + d \cdot D \cdot (N - 1)\right)\right).$$

The proof size is the same as that described in equation 50 but with $n = N^D$.

9.7.11 *MPCitHead-Verify-2*$((A, \mathbf{s}), prf)$

Again, verification takes roughly the same time as the proof generation, thus the complexity is

$$O\left(B \cdot \left(N^D + d \cdot D \cdot (N - 1)\right)\right).$$

9.7.12 *XOF-Share*(x, XOF)

This clearly has complexity $O(t \cdot n)$.

9.7.13 *MPCitHead-Prove-3*$((A, \mathbf{s}), \mathbf{b})$

Proof generation has time complexity dominated by

$$O(d \cdot B \cdot \tau \cdot t).$$

9.7.14 *MPCitHead-Verify-3*$((A, \mathbf{s}), prf)$

Once again, verification takes roughly the same time as the proof generation, thus the complexity is

$$O(d \cdot B \cdot \tau \cdot t).$$

The proof size for this third set of MPC–in–the–Head proofs is given by

$$sz \leq |\mathcal{H}_2| + |\mathcal{H}_4| + \tau \cdot sec \cdot \log_2 \left(\frac{2^{\lceil \log_2 M \rceil}}{\tau} \right)$$

$$+ \tau \cdot (B + t \cdot sec + B \cdot \log_2 q + (n - t) \cdot |Commit|)$$

$$= 4 \cdot sec + \tau \cdot sec \cdot \log_2 \left(\frac{2^{\lceil \log_2 M \rceil}}{\tau} \right)$$

$$+ \tau \cdot (B + t \cdot sec + B \cdot \log_2 q + 2 \cdot (n - t) \cdot sec)$$

$$= sec \cdot \left(4 + \tau \cdot \left(2 \cdot n - t + \log_2 \left(\frac{2^{\lceil \log_2 M \rceil}}{\tau} \right) \right) \right)$$

$$+ \tau \cdot B \cdot (1 + \log_2 q)$$

Note that, the proof size depends on the parameter q, unlike the prior proof.

9.7.15 Layer Five Summary

We summarize the above discussion in the following table:

Protocol	Rounds	Complexities Communication	Computation
CRS-Gen.Init$(sec, \bar{q}, \bar{d}, \bar{B})$	–	–	–
CRS-Gen.Update(pp_{j-1})	1	$O(k \cdot \bar{B}_1 \cdot \log_2 t)$	$2 \cdot \bar{B}_1 \cdot \mathbb{G} + \bar{B}_1 \cdot \hat{\mathbb{G}}$
CRS-Gen.Output()	–	–	–
VC-Prove-1$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$	–	–	$3 \cdot \bar{B}_1 \cdot \mathbb{G}$
VC-Verify-1$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), prf)$	–	–	$\bar{B}_1 \cdot (\mathbb{G} + \hat{\mathbb{G}}) + \mathbf{PP}$
VC-Prove-2$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), \mathbf{b})$	–	–	$(6 \cdot \bar{B}_1 + 2 \cdot N) \cdot \mathbb{G} + (\bar{B}_1 + 3 \cdot N) \cdot \hat{\mathbb{G}}$
VC-Verify-2$(\mathbf{p}, (\mathbf{A}, \mathbf{s}), prf)$	–	–	$128 \cdot \hat{\mathbb{G}} + 8 \cdot \mathbf{PP}$
MPCitHead-Prove-1$((A, \mathbf{s}), \mathbf{b})$	–	–	$O(d \cdot B \cdot \tau \cdot (n - 1))$
MPCitHead-Verify-1$((A, \mathbf{s}), prf)$	–	–	$O(d \cdot B \cdot \tau \cdot (n - 1))$
MPCitHead-Prove-2$((A, \mathbf{s}), \mathbf{b})$	–	–	$O\left(B \cdot \left(N^D + d \cdot D \cdot (N - 1)\right)\right)$
MPCitHead-Verify-2$((A, \mathbf{s}), prf)$	–	–	$O\left(B \cdot \left(N^D + d \cdot D \cdot (N - 1)\right)\right)$
XOF-Share(x, XOF)	–	–	$O(t \cdot n)$
MPCitHead-Prove-3$((A, \mathbf{s}), \mathbf{b})$	–	–	$O(d \cdot B \cdot \tau \cdot t)$
MPCitHead-Verify-3$((A, \mathbf{s}), prf)$	–	–	$O(d \cdot B \cdot \tau \cdot t)$

Table 19: Complexity of Layer Five Algorithms and Protocols.

10 Notation

10.1 Acronyms

Term	Meaning
ABB	Arithmetic Black Box
AES	Advanced Encryption Standard
A–w–A	Active–with–Abort
API	Application Programming Interface
BCH	Bose–Chaudhuri–Hocquenghem code
BFV	Brakerski–Fan–Vercauteren FHE Scheme
BGV	Brakerski–Gentry–Vaikuntanathan FHE Scheme
BGW	Ben–Or–Goldwasser–Wigderson MPC Protocol
CCD	Chaum–Crépeau–Damgård MPC Protocol
CRT	Chinese Remainder Theorem
DCRT	Double CRT Representation
FFT	Fast Fourier Transform
FHE	Fully Homomorphic Encryption
F–GLWE	Flattened GLWE
GLWE	Generalized Learning–With–Errors
GMW	Goldreich–Micali–Wigderson MPC Protocol
KEM	Key Encapsulation Mechanism
LAN	Local Area Network
LWE	Learning–With–Errors
MAC	Message Authentication Code
MPC	Multi–Party Computation
MK–FHE	Multi–Key Fully Homomorphic Encryption
MP–FHE	Multi–Party Fully Homomorphic Encryption
NTT	Number Theoretic Transform
PBS	Programmable Bootstrapping
PRF	Pseudo–Random Function
PRG	Pseudo–Random Generator
PKI	Public Key Infrastructure
PRSS	Pseudo–Random Secret Sharing
PRZS	Pseudo–Random Zero Sharing
ROM	Random Oracle Model
SFE	Secure Function Evaluation
SPDZ	Smart–Pastro–Damgård–Zakarias Protocol
TFHE	Torus Fully Homomorphic Encryption Scheme
TLS	Transport Layer Security Protocol
UC	Universal Composability
VSS	Verifiable Secret Sharing
WAN	Wide Area Network
XOF	Extendable Output Function
ZKPoK	Zero–Knowledge Proof–of–Knowledge

Table 20: Table of Acronyms.

10.2 Mathematical Symbols

Term	Meaning	See Page
sec	A computational security parameter (usually set to 128)	23
$dist$	A statistical distance security parameter (usually set to 80)	23
$stat$	A statistical security parameter related to masking (usually set to 40)	23
$\mathfrak{n}$	Number of players in our protocols	23
$\mathfrak{t}$	Maximum number of players who can be corrupted	23
$\binom{\mathfrak{n}}{\mathfrak{t}}$	Number of combinations of $\mathfrak{t}$ items selected from $\mathfrak{n}$ things	27
$q = p^k$ or $p_1 \cdots p_k$	The modulus for the Galois Ring we require	27
Q	LWE ciphertext modulus	32
P	LWE plaintext modulus	32
$NewHope(N, B)$	The NewHope distribution on $[-B, B]^N$	33
$TUniform(N, A, B)$	An approximately uniform distribution on $[A, B]^N$	35
$\leftarrow$	Assignment	22
$\mathbb{Z}_p$	Ring of p–adic integers	22
$\mathbb{Z}/(m)$	Ring of integers modulo m	22
$nSmall$	The profiles $\{nSfH, nmfH\}$	27
$nLarge$	The profiles $\{nMfH^+, nLfH^+, nEfH^+\}$	27
$nSmallBnd$	The division between the profiles $nSmall$ and $nLarge$	28
N	Ring–LWE ring dimension	51
$\mathbf{R}^{(M)}$	Global (non–reduced) LWE cyclotomic ring of degree $N = \phi(M)$	51
$\mathbf{R}_Q^{(M)}$	The ring $\mathbf{R}^{(M)}$ localized at Q	51
$\mathbf{R}_Q$	Abuse of notation for $\mathbf{R}_Q^{(M)}$ when M is implied by the context	51
$\mathbf{a} \cdot \mathbf{b}$	Dot product of two vectors	51
$\mathbf{a} \odot \mathbf{b}$	Ring product of two ring elements	51
$\Delta = \lfloor Q \rfloor P$	Scaling factor for BFV/TFHE	51
$\overleftrightarrow{\mathbf{a}}$	The reverse of a vector	52
$erfc$	The complimentary error function	56
$can(\mathbf{a})$	Canonical embedding of a ring element $\mathbf{a} \in \mathbf{R}$	56
$\|\mathbf{a}\|_\infty$	Polynomial basis norm of $\mathbf{a} \in \mathbf{R}$	56
$\|\mathbf{a}\|^{can}$	Canonical basis norm of $\mathbf{a} \in \mathbf{R}$	56
E_M	The ring constant of $\mathbf{R}_M$	56
$c_{err,N}$	The error constant for applying $erfc$ in a ring of dimension N	57
$K_n(z)$	The modified Bessel function of the second kind	58
B_{Scale}	Noise increase due to Type–I modulus switching	62
$\sigma_{\mathfrak{s}\mathfrak{k}}$	BGV secret key standard deviation	63
$B_{KeySwitch}$	BGV key switch noise bound	66
B_{Mult}	BGV noise management constant after multiplications	66
$nSfH$	A NIST defined threshold profile	99
$nMfH$	A NIST defined threshold profile	99
$nLfH$	A NIST defined threshold profile	99
$nEfH$	A NIST defined threshold profile	99
$nmfH$	A modified threshold profile	99
$nLfH^+$	A modified threshold profile	99
$nEfH^+$	A modified threshold profile	99
$nMfH^+$	A modified threshold profile	99

Table 21: Table of Mathematical Symbols.

10.3 Algorithms and Protocols

Table 22: Table of Algorithms and Protocols.

11 Acknowledgements

References

[ABD16]   Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions – cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 153–178, Santa Barbara, CA, USA, August 14–18, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53018-4_6.

[ABMP24]   Andreea Alexandru, Ahmad Al Badawi, Daniele Micciancio, and Yuriy Polyakov. Application-aware approximate homomorphic encryption: Configuring FHE for practical use. Cryptology ePrint Archive, Paper 2024/203, 2024. URL: https://eprint.iacr.org/2024/203.

[ABO23]   Bar Alon, Amos Beimel, and Eran Omri. Three party secure computation with friends and foes. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023: 21st Theory of Cryptography Conference, Part II*, volume 14370 of *Lecture Notes in Computer Science*, pages 156–185, Taipei, Taiwan, November 29 – December 2, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-48618-0_6.

[ACD+19]   Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-36030-6_19.

[ADPS16]   Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[AGH+23]   Carlos Aguilar-Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The return of the SDitH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 564–596, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-30589-4_20.

[AHIV17]   Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. doi:10.1145/3133956.3134104.

[AHIV23]   Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. *Designs, Codes and Cryptography*, 91(11):3379–3424, 2023. doi:10.1007/s10623-023-01222-8.

[AJL+12]   Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501, Cambridge, UK, April 15–19, 2012. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-29011-4_29.

[AJW11]   Gilad Asharov, Abhishek Jain, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. Cryptology ePrint Archive, Report 2011/613, 2011. URL: https://eprint.iacr.org/2011/613.

[AKÖ23]   Yavuz Akin, Jakub Klemsa, and Melek Önen. A practical TFHE-based multi-key homomorphic encryption with linear complexity and low noise growth. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *ESORICS 2023: 28th European Symposium on Research in Computer Security, Part I*, volume 14344 of *Lecture Notes in Computer Science*, pages 3–23, The Hague, The Netherlands, September 25–29, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-50594-2_1.

[AOP20]   Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with friends and foes. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of

Lecture Notes in Computer Science, pages 677–706, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-56880-1_24`.

[AP13]   Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20, Santa Barbara, CA, USA, August 18–22, 2013. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-642-40041-4_1`.

[APS15]   Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9:169–203, 2015. `doi:10.1515/jmc-2015-0016`.

[BBB+18]   Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press. `doi:10.1109/SP.2018.00020`.

[BBB+23]   Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *Journal of Cryptology*, 36(3):28, July 2023. `doi:10.1007/s00145-023-09463-5`.

[BBD+23]   Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part V*, volume 14085 of *Lecture Notes in Computer Science*, pages 581–615, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland. `doi:10.1007/978-3-031-38554-4_19`.

[BBG+21]   Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 129–161, Raleigh, NC, USA, November 8–11, 2021. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-90453-1_5`.

[BCG93]   Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th Annual ACM Symposium on Theory of Computing*, pages 52–61, San Diego, CA, USA, May 16–18, 1993. ACM Press. `doi:10.1145/167088.167109`.

[BCS19]   Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302, Waterloo, ON, Canada, August 12–16, 2019. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-38471-5_12`.

[BD10]   Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 201–218, Zurich, Switzerland, February 9–11, 2010. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-642-11799-2_13`.

[BdBB+25]   Mathieu Ballandras, Mayeul de Bellabre, Loris Bergerat, Charlotte Bonte, Carl Bootland, Benjamin R. Curtis, Jad Khatib, Jakub Klemsa, Arthur Meyre, Thomas Montaigu, Jean-Baptiste Orfila, Nicolas Sarlin, Samuel Tap, and David Testé. TFHE-rs: A (Practical) Handbook, 2025. URL: `https://github.com/zama-ai/tfhe-rs-handbook/blob/main/tfhe-rs-handbook.pdf`.

[BDO+21]   Karim Baghery, Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Nigel P. Smart, and Titouan Tanguy. Compilation of function representations for secure computing paradigms. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 26–50, Virtual Event, May 17–20, 2021. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-75539-3_2`.

[Bea92]   Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer Berlin Heidelberg, Germany. `doi:10.1007/3-540-46766-1_34`.

[Bea95]   Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109, Santa Barbara, CA, USA, August 27–31, 1995. Springer Berlin Heidelberg, Germany. `doi:10.1007/3-540-44750-4_8`.

[Bea97]   Donald Beaver. Commodity-based cryptography (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 446–455, El Paso, TX, USA, May 4–6, 1997. ACM Press. `doi:10.1145/258533.258637`.

[BEL+24]   Olivier Bernard, Sarah Elkazdadi, Benoît Libert, Arthur Meyre, Arthur Meyre, Jean-Baptiste Orfila, and Nicolas Sarlin. Faster Short Pairing-Based NIZK Proofs for Ring LWE Ciphertexts. In submission,

October 2024.

[BFH+20] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 2025–2038, Virtual Event, USA, November 9–13, 2020. ACM Press. doi:10.1145/3372297.3417893.

[BGG+18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 565–596, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-96884-1_19.

[BGIN19] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 869–886, London, UK, November 11–15, 2019. ACM Press. doi:10.1145/3319535.3363227.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery. doi:10.1145/2090236.2090262.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press. doi:10.1145/62212.62213.

[BHP17] Zvika Brakerski, Shai Halevi, and Antigoni Polychroniadou. Four round secure computation without setup. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 645–677, Baltimore, MD, USA, November 12–15, 2017. Springer, Cham, Switzerland. doi:10.1007/978-3-319-70500-2_22.

[BJSW24] Olivier Bernard, Marc Joye, Nigel P. Smart, and Michael Walter. Drifting towards better error probabilities in fully homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2024/1718, 2024. URL: https://eprint.iacr.org/2024/1718.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In Jim Anderson and Sam Toueg, editors, *13th ACM Symposium Annual on Principles of Distributed Computing*, pages 183–192, Los Angeles, CA, USA, August 14–17, 1994. Association for Computing Machinery. doi:10.1145/197917.198088.

[BLLN13] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *14th IMA International Conference on Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64, Oxford, UK, December 17–19, 2013. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-45239-0_4.

[BLR+18] Shi Bai, Tancrède Lepoint, Adeline Roux-Langlois, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. *Journal of Cryptology*, 31(2):610–640, April 2018. doi:10.1007/s00145-017-9265-9.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-45682-1_30.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-96878-0_17.

[BP16] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multi-key FHE with short ciphertexts. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 190–213, Santa Barbara, CA, USA, August 14–18, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53018-4_8.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417

of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-32009-5_50.

[BS23]   Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 371–404, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore. doi:10.1007/978-981-99-8721-4_12.

[BSS99]   Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999. doi:10.1017/CBO9781107360211.

[BTH08]   Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, San Francisco, CA, USA, March 19–21, 2008. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-78524-8_13.

[BV98]   Dan Boneh and Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 59–71, Espoo, Finland, May 31 – June 4, 1998. Springer Berlin Heidelberg, Germany. doi:10.1007/BFb0054117.

[BW86]   Elwyn R. Berlekamp and Lloyd R. Welch. Error correction of algebraic block codes. US Patent, Number 4,633,470, 1986. URL: https://patents.google.com/patent/US4633470A/en.

[Cas86]   J. W. S. Cassels. *Local Fields*. London Mathematical Society Student Texts. Cambridge University Press, 1986. doi:10.1017/CBO9781139171885.

[CCD88]   David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press. doi:10.1145/62212.62214.

[CCP22]   Anirudh Chandramouli, Ashish Choudhury, and Arpita Patra. A survey on perfectly secure verifiable secret-sharing. *ACM Comput. Surv.*, 54(11s):232:1–232:36, 2022. doi:10.1145/3512344.

[CCP+24]   Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. Attacks against the IND-CPA^D security of exact FHE schemes. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024: 31st Conference on Computer and Communications Security*, pages 2505–2519, Salt Lake City, UT, USA, October 14–18, 2024. ACM Press. doi:10.1145/3658644.3690341.

[CCS19]   Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from TFHE. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 446–472, Kobe, Japan, December 8–12, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-34621-8_16.

[CDE+18]   Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-96881-0_26.

[CDG+17]   Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1825–1842, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. doi:10.1145/3133956.3133997.

[CdH10]   Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199, Amalfi, Italy, September 13–15, 2010. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-15317-4_13.

[CDI05]   Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-30576-7_19.

[CDKS19]   Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 395–412, London, UK, November 11–15, 2019. ACM Press. doi:10.1145/3319535.3363207.

[CGGI16]   Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors,

Advances in Cryptology – ASIACRYPT 2016, Part I, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53887-6_1.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-96878-0_2.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *Fast Software Encryption – FSE 2015*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149, Istanbul, Turkey, March 8–11, 2015. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-48116-5_7.

[Che06] Jung Hee Cheon. Security analysis of the strong Diffie–Hellman problem. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 1–11, St. Petersburg, Russia, May 28 – June 1, 2006. Springer Berlin Heidelberg, Germany. doi:10.1007/11761679_1.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021. doi:10.1007/978-3-030-78086-9_1.

[CLO+13] Ashish Choudhury, Jake Loftus, Emmanuela Orsini, Arpita Patra, and Nigel P. Smart. Between a rock and a hard place: Interpolating between MPC and FHE. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 221–240, Bengalore, India, December 1–5, 2013. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-42045-0_12.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, December 6–10, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-92078-4_23.

[CM15] Michael Clear and Ciaran McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 630–656, Santa Barbara, CA, USA, August 16–20, 2015. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-48000-7_31.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, 1993. doi:10.1007/978-3-662-02945-9.

[Coh16] Ran Cohen. Asynchronous secure multiparty computation in constant time. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 183–207, Taipei, Taiwan, March 6–9, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-49387-8_8.

[CS16] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Kazue Sako, editor, *Topics in Cryptology – CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 325–340, San Francisco, CA, USA, February 29 – March 4, 2016. Springer, Cham, Switzerland. doi:10.1007/978-3-319-29485-8_19.

[CSBB24] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. On the practical CPA^D security of "exact" and threshold FHE schemes and libraries. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024, Part III*, volume 14922 of *Lecture Notes in Computer Science*, pages 3–33, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-68382-4_1.

[CSS+22] Siddhartha Chowdhury, Sayani Sinha, Animesh Singh, Shubham Mishra, Chandan Chaudhary, Sikhar Patranabis, Pratyay Mukherjee, Ayantika Chatterjee, and Debdeep Mukhopadhyay. Efficient threshold FHE with application to real-time systems. Cryptology ePrint Archive, Report 2022/1625, 2022. URL: https://eprint.iacr.org/2022/1625.

[CsW19]   Ran Cohen, abhi shelat, and Daniel Wichs. Adaptively secure MPC with sublinear communication complexity. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 30–60, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-26951-7_2`.

[CZW17]   Long Chen, Zhenfeng Zhang, and Xueqing Wang. Batched multi-hop multi-key FHE from ring-LWE with compact ciphertext extension. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 597–627, Baltimore, MD, USA, November 12–15, 2017. Springer, Cham, Switzerland. `doi:10.1007/978-3-319-70503-3_20`.

[DDE+23] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-FHE using noise flooding. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 35–46. ACM, 2023. `doi:10.1145/3605759.3625259`.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304, New York, NY, USA, March 4–7, 2006. Springer Berlin Heidelberg, Germany. `doi:10.1007/11681878_15`.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-642-00468-1_10`.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-642-40203-6_1`.

[DN07]   Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-540-74143-5_32`.

[dPLS19] Rafaël del Pino, Vadim Lyubashevsky, and Gregor Seiler. Short discrete log proofs for FHE and ring-LWE ciphertexts. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 11442 of *Lecture Notes in Computer Science*, pages 344–373, Beijing, China, April 14–17, 2019. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-17253-4_12`.

[DPR16]   Ivan Damgård, Antigoni Polychroniadou, and Vanishree Rao. Adaptively secure multi-party computation from LWE (via equivocal FHE). In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 208–233, Taipei, Taiwan, March 6–9, 2016. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-662-49387-8_9`.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. `doi:10.1007/978-3-642-32009-5_38`.

[DS83]   Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983. `doi:10.1137/0212045`.

[DVV19]   Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on ring/mod-LWE/LWR based schemes. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography – 10th International Conference, PQCrypto 2019*, pages 103–115, Chongqing, China, May 8–10, 2019. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-25510-7_6`.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland. `doi:10.1007/978-3-030-56880-1_29`.

[EHL+23] Daniel Escudero, Cheng Hong, Hongqing Liu, Chaoping Xing, and Chen Yuan. Degree-D reverse multiplication-friendly embeddings: Constructions and applications. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 106–138, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore. doi:10.1007/978-981-99-8721-4_4.

[Feh98] Serge Fehr. Span programs over rings and how to share a secret from a module. Master's thesis, ETH Zurich, Institute for Theoretical Computer Science, 1998. URL: https://crypto.ethz.ch/publications/Fehr98.html.

[FHM99] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. General adversaries in unconditional multi-party computation. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology – ASIACRYPT'99*, volume 1716 of *Lecture Notes in Computer Science*, pages 232–246, Singapore, November 14–18, 1999. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-48000-6_19.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-96881-0_2.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982. doi:10.1016/0020-0190(82)90033-3.

[FMRV22] Thibauld Feneuil, Jules Maire, Matthieu Rivain, and Damien Vergnaud. Zero-knowledge protocols for the subset sum problem from MPC-in-the-head with rejection. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 371–402, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-031-22966-4_13.

[FR23] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 441–473, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore. doi:10.1007/978-981-99-8721-4_14.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. URL: https://eprint.iacr.org/2012/144.

[Gao03] Shuhong Gao. *A New Algorithm for Decoding Reed-Solomon Codes*, pages 55–68. Springer US, Boston, MA, 2003. doi:10.1007/978-1-4757-3789-9_5.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. URL: https://crypto.stanford.edu/craig.

[GGHAK22] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Efficient set membership proofs using MPC-in-the-head. *Proceedings on Privacy Enhancing Technologies*, 2022(2):304–324, April 2022. doi:10.2478/popets-2022-0047.

[GGHR14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In Yehuda Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 74–94, San Diego, CA, USA, February 24–26, 2014. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-54242-8_4.

[GHL22] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 458–487, Trondheim, Norway, May 30 – June 3, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-031-06944-4_16.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-32009-5_49.

[GIKR01] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. The round complexity of verifiable secret sharing and secure multicast. In *33rd Annual ACM Symposium on Theory of Computing*, pages 580–589, Crete, Greece, July 6–8, 2001. ACM Press. doi:10.1145/380752.380853.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-26951-7_4.

[GMPP16] Sanjam Garg, Pratyay Mukherjee, Omkant Pandey, and Antigoni Polychroniadou. The exact round complexity of secure computation. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances*

      in Cryptology – EUROCRYPT 2016, Part II, volume 9666 of *Lecture Notes in Computer Science*, pages 448–476, Vienna, Austria, May 8–12, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-49896-5_16.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press. doi:10.1145/28395.28420.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326, Vienna, Austria, May 8–12, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-49896-5_11.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 618–646, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland. doi:10.1007/978-3-030-56880-1_22.

[GTLBNG21] José Gómez-Torrecillas, Francisco Javier Lobillo Borrero, and Gabriel Navarro Garulo. Decoding linear codes over chain rings given by parity check matrices. *Mathematics*, 9:1878, 2021. doi:10.3390/math9161878.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. URL: https://eprint.iacr.org/2019/953.

[Hal15] Jonathan I. Hall. Notes on coding theory: Chapter 5, 2015. URL: https://users.math.msu.edu/users/halljo/classes/codenotes/coding-notes.html.

[HKK+22] Aditya Hegde, Nishat Koti, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, and Protik Paul. Attaining GOD beyond honest majority with friends and foes. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part I*, volume 13791 of *Lecture Notes in Computer Science*, pages 556–587, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-031-22963-3_19.

[HKL21] David Heath, Vladimir Kolesnikov, and Jiahui Lu. Efficient generic arithmetic for KKW – practical linear mpc-in-the-head NIZK on commodity hardware without trusted setup. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning – 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 414–431. Springer, 2021. doi:10.1007/978-3-030-78086-9_31.

[HKLS24] Deokhwa Hong, Young-Sik Kim, Yongwoo Lee, and Eunyoung Seo. A new fine tuning method for FHEW/TFHE bootstrapping with IND-CPAD security. Cryptology ePrint Archive, Report 2024/1052, 2024. URL: https://eprint.iacr.org/2024/1052.

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482, Santa Barbara, CA, USA, August 20–24, 2006. Springer Berlin Heidelberg, Germany. doi:10.1007/11818175_28.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 322–340, Aarhus, Denmark, May 22–26, 2005. Springer Berlin Heidelberg, Germany. doi:10.1007/11426639_19.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. URL: https://eprint.iacr.org/2020/1481.

[Joy21] Marc Joye. Balanced non-adjacent forms. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 553–576, Singapore, December 6–10, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-92078-4_19.

[Joy24] Marc Joye. TFHE public-key encryption revisited. In Elisabeth Oswald, editor, *Topics in Cryptology – CT-RSA 2024*, volume 14643 of *Lecture Notes in Computer Science*, pages 277–291, San Francisco, CA, USA, May 6–9, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-58868-6_11.

[JSvL22] Robin Jadoul, Nigel P. Smart, and Barry van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021: 28th Annual International Workshop on Selected Areas in Cryptography*, volume 13203 of *Lecture Notes in Computer Science*, pages 131–151, Virtual Event, September 29 – October 1, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-030-99277-4_7.

[KKL+23] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 726–740, Copenhagen, Denmark, November 26–30, 2023. ACM Press. doi:10.1145/3576915.3623176.

[KKPG22] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. PentaGOD: Stepping beyond traditional GOD with five parties. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1843–1856, Los Angeles, CA, USA, November 7–11, 2022. ACM Press. doi:10.1145/3548606.3559369.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi:10.1145/3243734.3243805.

[KLP18] Eunkyung Kim, Hyang-Sook Lee, and Jeongeun Park. Towards round-optimal secure multiparty computations: Multikey FHE without a CRS. In Willy Susilo and Guomin Yang, editors, *ACISP 18: 23rd Australasian Conference on Information Security and Privacy*, volume 10946 of *Lecture Notes in Computer Science*, pages 101–113, Wollongong, NSW, Australia, July 11–13, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-93638-3_7.

[KMS24] Hyesun Kwak, Seonhong Min, and Yongsoo Song. Towards practical multi-key TFHE: Parallelizable, key-compatible, quasi-linear complexity. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part IV*, volume 14604 of *Lecture Notes in Computer Science*, pages 354–385, Sydney, NSW, Australia, April 15–17, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-57728-4_12.

[KMSV21] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 98–127, Singapore, December 6–10, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-92078-4_4.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press. doi:10.1145/2976749.2978357.

[KS73] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C22(8):786–793, 1973. doi:10.1109/TC.1973.5009159.

[Lib24] Benoît Libert. Vector commitments with proofs of smallness: Short range proofs and more. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 36–67, Sydney, NSW, Australia, April 15–17, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-57722-2_2.

[LM21] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 648–677, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-77870-5_23.

[LNS21] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Shorter lattice-based zero-knowledge proofs via one-time commitments. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 215–241, Virtual Event, May 10–13, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-75245-3_9.

[LP19] Hyang-Sook Lee and Jeongeun Park. On the security of multikey homomorphic encryption. In Martin Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *Lecture Notes in Computer Science*, pages 236–251, Oxford, UK, December 16–18, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-35199-1_12.

[LRY16] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP 2016: 43rd International Colloquium on Automata, Languages and Programming*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:14, Rome, Italy, July 11–15, 2016. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2016.30.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015. doi:10.1007/s10623-014-9938-4.

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *44th Annual ACM Symposium on Theory of Computing*, pages 1219–1234, New York, NY, USA, May 19–22, 2012. ACM Press. doi:10.1145/2213977.2214086.

[LY10] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517, Zurich, Switzerland, February 9–11, 2010. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-642-11799-2_30.

[MF21] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles*. Information Security and Cryptography. Springer, 2021. doi:10.1007/978-3-030-63287-8.

[MRY23] Nikolas Melissaris, Divya Ravi, and Sophia Yakoubov. Threshold-optimal MPC with friends and foes. In Anupam Chattopadhyay, Shivam Bhasin, Stjepan Picek, and Chester Rebeiro, editors, *Progress in Cryptology – INDOCRYPT 2023: 24th International Conference in Cryptology in India, Part II*, volume 14460 of *Lecture Notes in Computer Science*, pages 3–24, Goa, India, December 10–13, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-56235-8_1.

[MS23] Daniele Micciancio and Adam Suhl. Simulation-secure threshold PKE from LWE with polynomial modulus. Cryptology ePrint Archive, Report 2023/1728, 2023. URL: https://eprint.iacr.org/2023/1728.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763, Vienna, Austria, May 8–12, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-49896-5_26.

[NNL01] Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and tracing schemes for stateless receivers. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 41–62, Santa Barbara, CA, USA, August 19–23, 2001. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-44647-8_3.

[NO07] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007: 10th International Conference on Theory and Practice of Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360, Beijing, China, April 16–20, 2007. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-71677-8_23.

[NRBB24] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. In Christina Pöpper and Lejla Batina, editors, *ACNS 24: 22nd International Conference on Applied Cryptography and Network Security, Part III*, volume 14585 of *Lecture Notes in Computer Science*, pages 105–134, Abu Dhabi, UAE, March 5–8, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-54776-8_5.

[OSV20] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 254–283, San Francisco, CA, USA, February 24–28, 2020. Springer, Cham, Switzerland. doi:10.1007/978-3-030-40186-3_12.

[PAA+19] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions.

[PS16] Chris Peikert and Sina Shiehian. Multi-key FHE from LWE, revisited. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 217–238, Beijing, China, October 31 – November 3, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53644-5_9.

[PV05] Pascal Paillier and Damien Vergnaud. Discrete-log-based signatures may not be equivalent to discrete log. In Bimal K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 1–20, Chennai, India, December 4–8, 2005. Springer Berlin Heidelberg, Germany. doi:10.1007/11593447_1.

[QBC13] Guillaume Quintin, Morgan Barbier, and Christophe Chabot. On generalized Reed-Solomon codes over commutative and noncommutative rings. *IEEE Trans. Inf. Theory*, 59(9):5882–5897, 2013. doi:10.1109/TIT.2013.2264797.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority

(extended abstract). In *21st Annual ACM Symposium on Theory of Computing*, pages 73–85, Seattle, WA, USA, May 15–17, 1989. ACM Press. doi:10.1145/73007.73014.

[RST+22] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *Journal of Cryptology*, 35(1):5, January 2022. doi:10.1007/s00145-021-09416-w.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – IN-DOCRYPT 2019: 20th International Conference in Cryptology in India*, volume 11898 of *Lecture Notes in Computer Science*, pages 227–249, Hyderabad, India, December 15–18, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-35423-7_12.

[RY07] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 228–245, Barcelona, Spain, May 20–24, 2007. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-72540-4_13.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252, Santa Barbara, CA, USA, August 20–24, 1990. Springer, New York, USA. doi:10.1007/0-387-34805-0_22.

[Sco19] Michael Scott. Pairing implementation revisited. Cryptology ePrint Archive, Report 2019/077, 2019. URL: https://eprint.iacr.org/2019/077.

[sec09] secureSCM EU Project. Deliverable D9.2: Security Analysis, 2009. URL: https://faui1-files.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-69053-0_18.

[Sho05] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005. doi:10.1017/CBO9781139165464.

[Sma23] Nigel P. Smart. Practical and efficient FHE-based MPC. In Elizabeth A. Quaglia, editor, *19th IMA International Conference on Cryptography and Coding*, volume 14421 of *Lecture Notes in Computer Science*, pages 263–283, London, UK, December 12–14, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-47818-5_14.

[Tap23] Samuel Tap. *Constructing new tools for efficient homomorphic encryption*. PhD thesis, ENS Paris, 2023. URL: https://theses.hal.science/tel-04587370.

[YC17] Zhen-Hang. Yang and Yu-Ming Chu. On approximating the modified Bessel function of the second kind. *J. Inequal. Appl.*, 41, 2017. doi:10.1186/s13660-017-1317-z.

[YKHK18] Satoshi Yasuda, Yoshihiro Koseki, Ryo Hiromasa, and Yutaka Kawai. Multi-key homomorphic proxy re-encryption. In Liqun Chen, Mark Manulis, and Steve Schneider, editors, *ISC 2018: 21st International Conference on Information Security*, volume 11060 of *Lecture Notes in Computer Science*, pages 328–346, Guildford, UK, September 9–12, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-99136-8_18.

Appendix

Security Parameter Tables

These tables provide the values which were discussed in Section 5.2. They provide ball park figures for security of the LWE problem with the given secret key distribution, size of Q, the generalized LWE dimension $w \cdot N$ and with the noise distribution given by either by $TUniform(\cdot, -2^b, 2^b)$ or $NewHope(\cdot, B)$. There were computed with the **lwe–estimator** [APS15] update from December 16th 2024.

	$Q \approx 2^{512}$		$Q \approx 2^{768}$		$Q \approx 2^{1024}$		$Q \approx 2^{1536}$		$Q \approx 2^{2048}$	
B	$w \cdot N$	sec	$w \cdot N$	sec	$w \cdot N$	sec	$w \cdot N$	sec	$w \cdot N$	sec
1	20000	131.3	30000	132.1	40000	132.5	60000	133.3	80000	133.7
2	20000	131.5	30000	132.1	40000	132.6	60000	133.3	80000	133.8
3	20000	131.5	30000	132.2	40000	132.7	60000	133.3	80000	133.8
4	20000	131.5	30000	132.3	40000	132.7	60000	133.3	80000	133.8

Table 23: BGV/BFV Style Parameters. Secret key distribution $NewHope(N, 1)$, noise distribution $NewHope(w \cdot N, B)$, $w \cdot N$ a multiple of 5000, and large Q.

	$Q = 2^{64}$		$Q = 2^{128}$	
b	$w \cdot N$	sec	$w \cdot N$	sec
0	2626	130.11	5120	130.72
1	2595	130.10	5120	131.58
2	2559	130.09	5120	132.50
3	2520	130.08	5120	133.65
4	2481	130.06	5120	134.79
5	2442	130.05	5120	135.94
6	2403	130.04	5120	136.94
7	2363	130.32	4864	130.74
8	2328	131.87	4864	131.74
9	2290	130.14	4864	132.89
10	2251	130.12	4864	134.05
11	2212	130.11	4864	135.46
12	2173	130.10	4864	136.47
13	2133	130.08	4608	130.13
14	2094	130.07	4608	131.27
15	2055	130.05	4608	132.27
16	2016	130.04	4608	133.70
17	1977	130.03	4608	134.85
18	1941	132.70	4608	136.14
19	1902	130.13	4608	137.28
20	1862	130.11	4352	130.66
21	1823	130.09	4352	131.66
22	1784	130.08	4352	133.09
23	1744	130.06	4352	134.25
24	1705	130.04	4352	135.67

25	1666	130.02	4352	136.95
26	1630	130.01	4352	138.38
27	1593	130.26	4096	131.03
28	1554	130.24	4096	132.32
29	1514	130.22	4096	133.50
30	1475	130.19	4096	135.18
31	1435	130.17	4096	136.19
32	1396	130.15	4096	137.69
33	1359	130.21	3840	130.13
34	1319	130.19	3840	131.42
35	1279	130.17	3840	132.89
36	1240	130.16	3840	134.52
37	1200	130.13	3840	135.85
38	1163	130.22	3840	137.37
39	1123	130.19	3840	139.00
40	1083	130.17	3584	130.86
41	1044	130.14	3584	132.22
42	1004	130.12	3584	133.94
43	966	130.19	3584	135.38
44	926	130.16	3584	136.94
45	886	130.13	3584	138.66
46	848	130.26	3584	140.23
47	808	130.17	3328	131.16
48	768	130.13	3328	132.87
49	729	130.19	3328	134.60
50	–	–	3328	136.17
51	–	–	3328	138.17
52	–	–	3328	139.90
53	–	–	3072	130.39
54	–	–	3072	132.04
55	–	–	3072	133.96
56	–	–	3072	135.81
57	–	–	3072	137.61
58	–	–	3072	139.56
59	–	–	3072	141.60
60	–	–	2816	130.89
61	–	–	2816	132.93
62	–	–	2816	134.90
63	–	–	2816	137.03
64	–	–	2816	139.06
65	–	–	2816	141.34
66	–	–	2560	130.09
67	–	–	2560	131.95
68	–	–	2560	134.24
69	–	–	2560	136.38
70	–	–	2560	138.71
71	–	–	2560	141.11
72	–	–	2560	143.45
73	–	–	2304	130.79

74	–	–	2304	133.15
75	–	–	2304	135.59
76	–	–	2304	138.16
77	–	–	2304	140.60
78	–	–	2304	143.49
79	–	–	2304	146.18
80	–	–	2048	131.99
81	–	–	2048	134.78
82	–	–	2048	137.49
83	–	–	2048	140.49
84	–	–	2048	143.42
85	–	–	2048	146.65
86	–	–	1792	130.51
87	–	–	1792	133.41
88	–	–	1792	136.66
89	–	–	1792	139.96

Table 24: TFHE Style Parameters. Secret key distribution $\{0, 1\}^N$, noise distribution $TUniform(w \cdot N, -2^b, 2^b)$, restricted $w \cdot N$ to a multiple of 256 when $Q = 2^{128}$.

B A Modified Threshold Key Generation

In the main body we comment that the method for threshold key generation presented requires an offline phase which needs to produce a large number of secret shared random bit values, which in turn require us to produce a large number of multiplication triples. This is because, due to Design Decision 22 we aim to produce FHE keys for use in our threshold protocol have the same properties/sizes as in non–threshold protocols. Thus the FHE parameters should not depend on either $\mathfrak{n}$ or $\mathfrak{t}$.

In this Appendix we present a half–way–house approach in which the FHE parameters depend slightly on the values of $\mathfrak{n}$ and $\mathfrak{t}$, but the number of multiplication triples required to be produced in the offline phase are smaller. This method only works for threshold profiles *nSmall* as it requires access to the operation *PRSS-Mask.Next*.

Recall *PRSS-Mask.Next*($Bd, stat$) produces the secret sharing of a sum of uniformly random values in the range $[-2^{stat} \cdot Bd, \ldots, 2^{stat} \cdot Bd]$. It was used previously to statistically mask a value of size bounded by Bd. The sum consists of $\binom{\mathfrak{n}}{\mathfrak{t}}$ terms, and during the execution of *PRSS-Mask.Next*($Bd, stat$) adversary can learn up to all bar one of the random values making up the sum. Write the shared value produced by a call to *PRSS-Mask.Next*($Bd, stat$) as $e_h + e_a$, where e_h is a value unknown to the adversary and e_a is the part of the sum known by the adversary.

However, if we call *PRSS-Mask.Next*($Bd, 0$) then we can treat the output as the noise term for an LWE sample, i.e.

$$(\mathbf{a}, b = \mathbf{a} \cdot \mathbf{s} + e)$$

where e is the underlying value secret shared by the call to *PRSS-Mask.Next*($Bd, 0$). The size of the noise sample is bounded by

$$|e_h + e_a| \le \binom{\mathfrak{n}}{\mathfrak{t}} \cdot Bd.$$

252

On one hand it has standard deviation

$$\sigma_{Bd} = \sqrt{\binom{\mathfrak{n}}{\mathfrak{t}}} \cdot \frac{2 \cdot Bd}{\sqrt{12}} = \sqrt{\binom{\mathfrak{n}}{\mathfrak{t}}/3} \cdot Bd.$$

On the other hand the LWE security is only achieved from the unknown value e_h and not the sum $e_h + e_a$. One can think of using such a method to generate the LWE sample as sampling $e_h + e_a$ as the LWE noise to produce the LWE sample,

$$(\mathbf{a}, b = \mathbf{a} \cdot \mathbf{s} + e_h + e_a)$$

and then leaking the value e_a to the adversary. Thus the adversary obtains the LWE sample

$$(\mathbf{a}, b' = \mathbf{a} \cdot \mathbf{s} + e_h).$$

In the worst case we can assume there is only one uniformly random value in the sum e_h, and so our LWE security depends solely on the size of Bd.

Thus whilst we can sample the noise using a fixed parameter Bd, the noise analysis (which produces the final FHE parameters) will depend on the standard deviation, i.e. it will depend on $\binom{\mathfrak{n}}{\mathfrak{t}}$. The advantage though is that the random bits, needed to produce the noise values in our previous methods, are replaced by a non–interactive call to *PRSS-Mask.Next*. We still require random bits to produce the secret key distribution, but these are a relatively small number of bits.

This results in the following algorithmic changes, which we outline in Figure 115, Figure 116, and Figure 117. The main changes are as follows:

- **BGV:** The change is in line 7 and 9, where we replace a call to *NewHope* by a call to *PRSS-Mask.Next*. Random bits are still required in line 1, in order to generate the secret key.
- **BFV:** Exactly the same change is applied in this algorithm as was done for BGV.
- **TFHE:** We replace calls to *MPC.TUniform* with calls to *PRSS-Mask.Next* in three places. In line 2 of *MPC.Enc^{LWE}*, in line 2 of *MPC.Enc^{GLWE}* and in line 9 of *TFHE.Threshold-KeyGen-2*.

The concrete complexities, from Section 9.6.14, become, with the other formulae remaining the same,

$$|BGV.Threshold\text{-}KeyGen\text{-}2(N, Q, P, B, R)|_T = |MPC.NewHope(N, 1)|_T$$
$$+ N \cdot |MPC.Mult(\cdot, \cdot)|_T,$$
$$|MPC.Enc^{GGSW}(\cdots, N, w, \cdot, \nu, \cdot)|_T = w \cdot N \cdot |MPC.Mult(\cdot, \cdot)|_T,$$
$$|TFHE.Threshold\text{-}KeyGen\text{-}2(\cdots, \hat{\ell}, \ell, N, w, \cdots,$$
$$\cdots, \nu_{pksk}, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \cdot, \overline{\nu_{bk}}, \cdot, \overline{flag})|_T = |MPC.GenBits(\ell)|_T$$
$$+ |MPC.GenBits(\hat{\ell})|_T$$
$$+ w \cdot |MPC.GenBits(N)|_T$$
$$+ \overline{flag} \cdot w \cdot |MPC.GenBits(\overline{N})|_T$$
$$+ \ell \cdot |MPC.Enc^{GGSW}(\cdots, N, w, \cdot, \nu_{bk}, \cdot)|_T$$
$$+ \ell \cdot \overline{flag} \cdot |MPC.Enc^{GGSW}(\cdots, \overline{N}, \overline{w}, \cdot, \overline{\nu_{bk}}, \cdot)|_T$$

BGV Threshold Key Generation – V2

$BGV.Threshold\text{-}KeyGen\text{-}2(N, Q, P, B, R)$:

Unless otherwise marked, all the secret sharings in this algorithm are modulo $q = T = Q \cdot R$.

1. $\langle \mathfrak{sk} \rangle \leftarrow MPC.NewHope(N, 1)$.
2. $\langle \mathfrak{pk}_a \rangle \leftarrow MPC^O.NextRandom()$.
3. $\langle \mathfrak{pk}'_a \rangle \leftarrow MPC^O.NextRandom()$.
4. $\mathfrak{pk}_a \leftarrow MPC.Open(\langle \mathfrak{pk}_a \rangle)$.
5. $\mathfrak{pk}'_a \leftarrow MPC.Open(\langle \mathfrak{pk}'_a \rangle)$.
6. $\mathfrak{pk}_a \leftarrow \mathfrak{pk}_a \pmod{Q}$.
7. $\langle \mathbf{e}_{\mathfrak{pk}} \rangle \leftarrow PRSS\text{-}Mask.Next(B, stat)$.
   This is in fact N calls to $PRSS\text{-}Mask.Next$, one per coefficient
8. $\langle \mathfrak{pk}_b \rangle \leftarrow \mathfrak{pk}_a \odot \langle \mathfrak{sk} \rangle + P \cdot \langle e_{\mathfrak{pk}} \rangle$.
9. $\langle \mathbf{e}'_{\mathfrak{pk}} \rangle \leftarrow PRSS\text{-}Mask.Next(B, stat)$.
   This is intact N calls to $PRSS\text{-}Mask.Next$, one per coefficient.
10. For $i, j \in [0, \ldots, N-1]$ do
    (a) $\langle s_{i,j} \rangle \leftarrow MPC.Mult(\langle \mathfrak{sk}_i \rangle, \langle \mathfrak{sk}_j \rangle)$.
11. $\langle \mathbf{s} \rangle \leftarrow \langle 0 \rangle$.
12. For $i, j \in [0, \ldots, N-1]$ do
    (a) If $i + j < N$ then $\langle s_{i+j} \rangle \leftarrow \langle s_{i+j} \rangle + \langle s_{i,j} \rangle$.
    (b) Else $\langle s_{i+j-N} \rangle \leftarrow \langle s_{i+j-N} \rangle - \langle s_{i,j} \rangle$.
13. $\langle \mathfrak{pk}'_b \rangle \leftarrow \mathfrak{pk}'_a \odot \langle \mathfrak{sk} \rangle + P \cdot \langle \mathbf{e}'_{\mathfrak{pk}} \rangle - R \cdot \langle \mathbf{s} \rangle$.
14. $\mathfrak{pk}_b \leftarrow MPC.Open(\langle \mathfrak{pk}_b \rangle)$.
15. $\mathfrak{pk}'_b \leftarrow MPC.Open(\langle \mathfrak{pk}'_b \rangle)$.
16. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_b \pmod{Q}$.
17. $\langle \mathfrak{sk} \rangle_{Q_1} \leftarrow \langle \mathfrak{sk} \rangle \pmod{Q_1}$, i.e. restrict $\langle \mathfrak{sk} \rangle$ to a secret sharing modulo Q_1.
18. $\mathfrak{pk} \leftarrow \{(\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}'_a, \mathfrak{pk}'_b)\}$.
19. Return $(\mathfrak{pk}, \langle \mathfrak{sk} \rangle_{Q_1})$.

Figure 115: BGV Threshold Key Generation V2, for Threshold Profile *nSmall*.

BFV Threshold Key Generation – V2

BFV.Threshold-KeyGen-2(N, Q, P, B, R):

Unless otherwise marked, all the secret sharings in this algorithm are modulo $q = T = Q \cdot R$.

1. $\langle \mathfrak{st} \rangle \leftarrow$ *MPC.NewHope*($N, 1$).
2. $\langle \mathfrak{pk}_a \rangle \leftarrow MPC^O.NextRandom()$.
3. $\langle \mathfrak{pk}'_a \rangle \leftarrow MPC^O.NextRandom()$.
4. $\mathfrak{pk}_a \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}_a \rangle$).
5. $\mathfrak{pk}'_a \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}'_a \rangle$).
6. $\mathfrak{pk}_a \leftarrow \mathfrak{pk}_a \pmod{Q}$.
7. $\langle \mathbf{e}_{\mathfrak{pk}} \rangle \leftarrow$ *PRSS-Mask.Next*($B, stat$).
   This is infant N calls to *PRSS-Mask.Next*, *one per coefficient*
8. $\langle \mathfrak{pk}_b \rangle \leftarrow \mathfrak{pk}_a \odot \langle \mathfrak{st} \rangle + \langle e_{\mathfrak{pk}} \rangle$.
9. $\langle \mathbf{e}'_{\mathfrak{pk}} \rangle \leftarrow$ *MPC.NewHope*(N, B).
10. $\langle \mathbf{e}'_{\mathfrak{pk}} \rangle \leftarrow$ *PRSS-Mask.Next*($B, stat$).
    This is infect N calls to *PRSS-Mask.Next*, *one per coefficient*
11. For $i, j \in [0, \ldots, N-1]$ do
    (a) $\langle s_{i,j} \rangle \leftarrow$ *MPC.Mult*($\langle \mathfrak{st}_i \rangle, \langle \mathfrak{st}_j \rangle$).
12. $\langle \mathbf{s} \rangle \leftarrow \langle 0 \rangle$.
13. For $i, j \in [0, \ldots, N-1]$ do
    (a) If $i + j < N$ then $\langle s_{i+j} \rangle \leftarrow \langle s_{i+j} \rangle + \langle s_{i,j} \rangle$.
    (b) Else $\langle s_{i+j-N} \rangle \leftarrow \langle s_{i+j-N} \rangle - \langle s_{i,j} \rangle$.
14. $\langle \mathfrak{pk}'_b \rangle \leftarrow \mathfrak{pk}'_a \odot \langle \mathfrak{st} \rangle + P \cdot \langle \mathbf{e}'_{\mathfrak{pk}} \rangle - R \cdot \langle \mathbf{s} \rangle$.
15. $\mathfrak{pk}_b \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}_b \rangle$).
16. $\mathfrak{pk}'_b \leftarrow$ *MPC.Open*($\langle \mathfrak{pk}'_b \rangle$).
17. $\mathfrak{pk}_b \leftarrow \mathfrak{pk}_b \pmod{Q}$.
18. $\langle \mathfrak{st} \rangle_{Q_1} \leftarrow \langle \mathfrak{st} \rangle \pmod{Q_1}$, i.e. restrict $\langle \mathfrak{st} \rangle$ to a secret sharing modulo Q_1.
19. $\mathfrak{pk} \leftarrow \{ (\mathfrak{pk}_a, \mathfrak{pk}_b), (\mathfrak{pk}'_a, \mathfrak{pk}'_b) \}$.
20. Return $(\mathfrak{pk}, \langle \mathfrak{st} \rangle_{Q_1})$.

Figure 116: BFV Threshold Key Generation V2, for Threshold Profile *nSmall*.

TFHE Threshold Key Generation – V2

$MPC.Enc^{LWE}(\langle m \rangle, \langle \mathbf{s} \rangle; XOF, P', Q, \ell, flag)$:
1. $\mathbf{\alpha} \leftarrow Expand^{LWE}(XOF, Q, \ell)$.
2. $\langle e \rangle \leftarrow PRSS\text{-}Mask.Next(2^{b_\ell}, stat)$.
3. $\langle b \rangle \leftarrow \mathbf{\alpha} \cdot \langle \mathbf{s} \rangle + \langle e \rangle + (Q/P') \cdot \langle m \rangle$.
4. If $flag$ return $(\mathbf{\alpha}, \langle b \rangle)$, else return $\langle b \rangle$.

$MPC.Enc^{GLWE}(\langle \mathbf{m} \rangle, (\langle \mathbf{s}_0 \rangle, \ldots, \langle \mathbf{s}_{w-1} \rangle); XOF, P', Q, N, w, flag)$:
1. $(\mathbf{\alpha}_0, \ldots, \mathbf{\alpha}_{w-1}) \leftarrow Expand^{GLWE}(XOF, Q, N, w)$.
2. $\langle \mathbf{e} \rangle \leftarrow PRSS\text{-}Mask.Next(2^{b_{w \cdot N}}, stat)$.
   This is infarct N calls to PRSS-Mask.Next, one per coefficient
3. $\langle \mathbf{b} \rangle \leftarrow \sum_{i=0}^{w-1} \mathbf{\alpha}_i \odot \langle \mathbf{s}_i \rangle + \langle \mathbf{e} \rangle + (Q/P') \cdot \langle \mathbf{m} \rangle \pmod{Q}$.
4. If $flag$ return $(\mathbf{\alpha}_0, \ldots, \mathbf{\alpha}_{w-1}, \langle \mathbf{b} \rangle)$, else return $\langle \mathbf{b} \rangle$.

$TFHE.Threshold\text{-}KeyGen\text{-}2(P, Q, flag, \hat{\ell}, \ell, N, w, \beta_{pksk}, \beta_{ksk}, \beta_{bk}, \nu_{pksk}, \nu_{ksk}, \nu_{bk}, \overline{N}, \overline{w}, \overline{Q}, \overline{\beta_{bk}},$
$\overline{\nu_{bk}}, flag, \overline{flag})$:
1. If $\overline{flag}$ then $\langle \cdot \rangle$ denotes secret sharing modulo $q = \overline{Q}$, otherwise it denotes secret sharing modulo $q = Q$.
2. For $i = 1, \ldots, \lceil sec/\log_2 q \rceil$ do
   (a) $\langle seed_i \rangle \leftarrow MPC^O.NextRandom()$.
   (b) $seed_i \leftarrow MPC.Open(\langle seed_i \rangle)$.
3. $seed \leftarrow (seed_1 \| \ldots \| seed_{\lceil sec/\log_2 q \rceil}) \pmod{2^{sec}}$.
4. $XOF.Init(seed, DSep(TFHE))$.
5. $\langle \hat{\mathbf{s}} \rangle \leftarrow MPC.GenBits(\hat{\ell})$.
6. $\langle \mathbf{s} \rangle \leftarrow MPC.GenBits(\ell)$.
7. For $i \in [0, \ldots, w-1]$ do $\langle \mathbf{s}_i \rangle \leftarrow MPC.GenBits(N)$.
8. $\mathfrak{pk}_a \leftarrow XOF.Next(\hat{\ell}, Q)$.
9. $\langle \mathbf{e} \rangle \leftarrow PRSS\text{-}Mask.Next(2^{b_i}, stat)$.
   This is infarct $\hat{\ell}$ calls to PRSS-Mask.Next, one per element of $\mathbf{e}$.
10. $\langle \overleftrightarrow{\mathfrak{pk}_b} \rangle \leftarrow \mathfrak{pk}_a \odot \overleftrightarrow{\langle \hat{\mathbf{s}} \rangle} + \langle \mathbf{e} \rangle$.
11. $\mathfrak{pk}_b \leftarrow MPC.Open(\langle \overleftrightarrow{\mathfrak{pk}_b} \rangle) \pmod{Q}$.
12. …
13. Return $(PK, \langle \mathfrak{sk} \rangle)$.

Figure 117: TFHE Threshold Key Generation V2, for Threshold Profile *nSmall*.

B.1 Changes to BGV/BFV Parameter Analysis

The change in above method of generating the noise for BGV and BFV changes the noise equations slightly. We basically need to change the standard deviation of $\sqrt{B \cdot N/2}$, used for the $NewHope(N, B)$ distribution in the original analysis, with the value $\sigma_B \cdot \sqrt{N}$. Thus the standard deviation increases by a factor of

$$\frac{\sqrt{\binom{\mathfrak{n}}{\mathfrak{t}}/3} \cdot B \cdot \sqrt{N}}{\sqrt{B \cdot N/2}} = \sqrt{2 \cdot \binom{\mathfrak{n}}{\mathfrak{t}} \cdot B/3},$$

assuming we use the same (minimal) value of $B = 1$ as chosen for BGV/BFV key generation earlier. In what follows we utilize the bound of $nSmallBnd$ on $\binom{\mathfrak{n}}{\mathfrak{t}}$ in order to bound

$$\sigma_B \leq \sqrt{nSmallBnd/3} \cdot B.$$

Our associated constants then become

$$B_{\mathfrak{ct}} = c_{err,N} \cdot P \cdot \left(\sqrt{N/12} + c_{err,N} \cdot \sigma_B \cdot \sqrt{N} \cdot \sqrt{N/2} + \sigma_B \cdot \sqrt{N} + c_{err,N} \cdot \sigma_B \cdot \sqrt{N} \cdot \sqrt{N/2} \right)$$

$$= c_{err,N} \cdot P \cdot \left(\sqrt{N/12} + \sigma_B \cdot \left(c_{err,N} \cdot \sqrt{2} \cdot N + \sqrt{N} \right) \right),$$

$$B_{KeySwitch} = c_{err,N}^2 \cdot P \cdot \sigma_B \cdot N/\sqrt{12}.$$

The value of B_{Scale} stays the same at

$$B_{Scale} = c_{err,N} \cdot (P + 1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sqrt{N/2})$$

In terms of setting parameters, the analysis changes as follows. We select R so that

$$R \approx \frac{256 \cdot B_{KeySwitch} \cdot Q}{B_{Scale}},$$

$$= \frac{256 \cdot Q \cdot c_{err,N}^2 \cdot P \cdot \sigma_B \cdot N/\sqrt{12}}{c_{err,N} \cdot (P + 1) \cdot \sqrt{N/12} \cdot (1 + c_{err,N} \cdot \sqrt{N/2})},$$

$$= 256 \cdot \sqrt{72} \cdot \sigma_B \cdot Q.$$

This gives us the parameters in Table 25, which also summarizes the parameters for the non–threshold version of BGV, our original threshold version from the main body, as well as the number of multiplication triples needed to perform key generation.

	Non–Threshold	Main Body Threshold	Appendix Threshold
λ	16	16	16
N	65536	65536	65536
B	1	1	1
λ	16	16	16
L	16	15	15
$B_{Mult} \approx$	$2^{37.49}$	$2^{37.49}$	$2^{37.49}$
$Q_\ell \approx (\ell \neq 1)$	$2^{46.56}$	$2^{46.56}$	$2^{46.56}$
$Q_1 \approx$	$2^{39.07}$	$2^{92.78}$	$2^{92.78}$
$Q \approx$	$2^{737.52}$	$2^{744.66}$	$2^{744.65}$
$R \approx$	$2^{745.52}$	$2^{752.66}$	$2^{761.59}$
$Q \cdot R \approx$	$2^{1483.04}$	$2^{1497.32}$	$2^{1506.25}$
$B_{Scale} \approx$	$2^{36.49}$	$2^{36.49}$	$2^{36.49}$
$B_{KeySwitch} \approx$	$2^{36.49}$	$2^{36.49}$	$2^{42.84}$

$B_{ct} \approx$	$2^{38.78}$	$2^{38.78}$	$2^{45.14}$
Number Triples	–	458,752	196,608

Table 25: Summary of all the parameters for BGV/BFV in this document for a plaintext modulus of $P = 65537$.

We see that the parameters change slightly, but not that much, with the number of triples reducing by over one half. These parameters are produced assuming $\binom{n}{t}$ can increase up to *nSmallBnd*, i.e. $10,000$. For smaller values of *nSmallBnd* the values in the final column will be closer to the values in the penultimate column.

B.2 Changes to TFHE Parameter Analysis

The noise analysis for the TFHE algorithm also changes as we move from using the *TUniform* distribution to generate noise via secret shared bits, to the non–interactive use of the *PRSS-Mask.Next* operation. The analysis of Section 5.6.3 carries over. The difference being that we replace

$$\sigma_{b_\ell} = \sigma_{ksk} = \sqrt{(2^{2 \cdot b_\ell + 1} + 1)/6},$$
$$\sigma_{bk} = \sqrt{(2^{2 \cdot b_{w \cdot N} + 1} + 1)/6},$$
$$\overline{\sigma_{bk}} = \sqrt{(2^{2 \cdot b_{\overline{w} \cdot \overline{N}} + 1} + 1)/6},$$

with

$$\sigma_{b_\ell} = \sigma_{ksk} = 2^{b_\ell} \cdot \sqrt{\binom{n}{t}/3},$$

$$\sigma_{bk} = 2^{b_{w \cdot N}} \cdot \sqrt{\binom{n}{t}/3},$$

$$\overline{\sigma_{bk}} = 2^{b_{\overline{w} \cdot \overline{N}}} \cdot \sqrt{\binom{n}{t}/3}.$$

However, this dependence of the σ values on $\binom{n}{t}$ means we need to decrease the maximum value of *nSmallBnd* from $10,000$ to 100. We need to do this for TFHE, as opposed to BGV/BFV, due to the limited space to accommodate noise due to the smaller value of the ciphertext modulus Q.

This gives us the parameters in Table 26, which also summarizes the parameters from the original threshold version from the main body, as well as the number of multiplication triples needed to perform threshold key generation. We are unable to find suitable parameters when $P = 32$ and $type = LWE$ due to the increased noise terms. As one can see, when we are able to find parameters, the number of required triples decreases significantly using the threshold key generation of this section, compared to that in the main body.

	Main Body Version				Appendix Version			
nSmallBnd	10,000				100			
	type = LWE		type = F-GLWE		type = LWE		type = F-GLWE	
	$P = 8$	$P = 32$	$P = 8$	$P = 32$	$P = 8$	$P = 32$	$P = 8$	$P = 32$
λ	2	5	2	5	2	–	2	5
$\hat{\ell}$	1024	2048	1024	2048	1024	–	1024	2048
ℓ	926	1004	848	926	1004	–	1004	1004
w	2	1	2	1	2	–	2	1
N	1024	4096	1024	4096	1024	–	1024	4096
Q	2^{64}	2^{64}	2^{64}	2^{64}	2^{64}	–	2^{64}	2^{64}
ν_{pksk}	2	4	1	1	3	–	1	1
β_{pksk}	2^7	2^4	2^{15}	2^{17}	2^5	–	2^{15}	2^{17}
ν_{bk}	1	1	1	1	1	–	1	1
β_{bk}	2^{18}	2^{21}	2^{18}	2^{22}	2^{18}	–	2^{18}	2^{22}
ν_{ks}	2	5	2	3	3	–	2	5

β_{ks}	2^7	2^4	2^6	2^5	2^5	–	2^6	2^3
$b_{\hat{\ell}}$	42	16	42	16	42	–	42	16
b_ℓ	44	42	46	44	42	–	42	42
$b_{w\cdot N}$	16	0	16	0	16	–	16	0
$\overline{w}$	4	1	4	1	4	–	4	1
$\overline{N}$	1024	4096	1024	4096	1024	–	1024	4096
$\overline{Q}$	2^{128}	2^{128}	2^{128}	2^{128}	2^{128}	–	2^{128}	2^{128}
$\overline{v_{bk}}$	3	3	3	3	3	–	3	3
$\overline{\beta_{bk}}$	2^{24}	2^{24}	2^{24}	2^{24}	2^{24}	–	2^{24}	2^{24}
$b_{\overline{w}\cdot\overline{N}}$	27	27	27	27	27	–	27	27
	Number of Triples							
$\overline{flag}=false$	53,432,222	21,867,500	48,891,728	19,577,758	2,060,268	–	2,060,268	4,119,532
$\overline{flag}=true$	469,706,654	741,538,796	430,102,352	683,338,654	6,176,748	–	6,176,748	8,236,012

Table 26: Summary of all the parameters for TFHE in this document.