# Trilithium: Efficient and Universally Composable Distributed ML-DSA Signing

Antonín Dufka[1[0009−0003−5058−2571]], Semjon Kravtšenko[1[0009−0007−8204−9519]], Peeter Laud[1[0000−0002−9030−8142]], and Nikita Snetkov[1,2[0000−0002−1414−2080]]

[1] Cybernetica AS, Mäealuse 2/1, 12618 Tallinn, Estonia
{antonin.dufka, semjon.kravtsenko, peeter.laud, nikita.snetkov}@cyber.ee
[2] Tallinn University of Technology, Akadeemia tee 15a, 12618 Tallinn

**Abstract.** In this paper, we present Trilithium: a protocol for distributed key generation and signing compliant with FIPS 204 (ML-DSA). Our protocol allows two parties, "server" and "phone" with assistance of correlated randomness provider (CRP) to produce a standard ML-DSA signature. We prove our protocol to be secure against a malicious server or phone in the universal composability (UC) model, introducing some novel techniques to argue the security of two-party secure computation protocols with active security against one party, but only active privacy against the other. We provide an implementation of our protocol in Rust and benchmark it, showing the practicality of the protocol.

**Keywords:** ML-DSA · Crystals-Dilithium · distributed signing · MPC · Universal Composability · threshold signatures

## 1 Introduction

In summer of 2022, National Institute of Standards and Technology (NIST) announced a selection of algorithms to become future cryptographic standards [3]. For key establishment algorithm (KEM) category Crystals-Kyber (ML-KEM) [8] was selected; and for digital signatures – Crystals-Dilithium (ML-DSA) [35], Falcon (FN-DSA) [39] and SPHINCS$^+$ (SLH-DSA) [5] were chosen. In August 2024, NIST published a a set of finalized standards, FIPS 203-205 [4]. All selected schemes are considered to be post-quantum secure (also known as *quantum-safe*) i.e. resilient from both classical and quantum adversarial attacks [5].

Parallel to the initiative on post-quantum cryptography, NIST published a draft of future call for multi-party threshold schemes [64]. The primary aim of this call is to construct advanced specifications for threshold schemes, including for NIST standardized and non-standardized cryptographic primitives. On top

[3] https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
[4] https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards
[5] https://www.etsi.org/technologies/quantum-safe-cryptography

of the interest from standardization agencies, distributed signatures are already being used in real-life applications such as digital signing services [6], distributed ledgers [7] and identity wallets [8] [9].

Over the past few years, substantial number of works have studied the topic of distributed signatures (both $t$-out-of-$n$ and $n$-out-of-$n$). For quantum-vulnerable schemes there exist efficient and practical distributed signing protocols: for RSA [43, 24, 11, 10, 19], ECDSA [41, 55, 30, 15, 29, 16, 13, 71, 53, 31, 49], probabilistic [62, 34, 60, 47, 54] and deterministic [61, 40, 37, 48] Schnorr signature schemes. There is also a fair amount of post-quantum distributed signature schemes [46, 17, 3, 22, 33], especially in the area of lattice-based protocols [26, 9, 38, 1, 42, 68, 44]. While many of them can be considered efficient, none of them are interoperable with new standards, except Cozzo and Smart [20] protocols. Still, their approach does not provide active security and directly could not be applied in real-life applications without substantial modifications.

Two-party protocols are an important use-case for threshold signing, having established a firm foothold in ecosystems with formalized frameworks of certification and assurance. Smart-ID [10] is a deployment of two-party RSA signatures, used by millions of residents of several European countries, and certified as a Qualified Signature Creation Device (QSCD) [11]. It is deployed as a central server, keeping one share of every user's signing key, while the second share is managed by that user's smartphone. It is interoperable with the rest of the ecosystem — the signatures created with the Smart-ID service are indistinguishable from any other RSA signatures and can be verified by anyone.

With the transition to quantum-safe primitives, currently deployed threshold schemes have to be replaced. While there are several pathways towards interoperability, certification-reliant ecosystems demand the use of standardized cryptosystems. As ML-DSA appears to be on its way to becoming the *default* postquantum-secure signature scheme [12], we ask: *Can an efficient and actively secure distributed protocol for ML-DSA be constructed?* In this paper, we give an affirmative answer to this question.

[6] https://www.smart-id.com/
[7] https://github.com/coinbase/cb-mpc
[8] https://irma.app/docs/getting-started/
[9] https://github.com/cleverbase/scal3
[10] https://www.smart-id.com/
[11] https://www.smart-id.com/e-service-providers/smart-id-as-a-qscd/
[12] https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

*Our Contributions* [13]

1. We propose *Trilithium* [14], a protocol for distributed ML-DSA key generation and signing, and prove it secure in the universal composability (UC) model.
   - We give a framework for actively *private* multiparty computation protocols in the UC model.
2. We provide an implementation and benchmarks of *Trilithium*.

### Related work

While there exist numerous constructions of lattice-based threshold signature schemes, as referenced above, our techniques are rather unrelated to them. Our protocol is an instance of classical secret-sharing [51] based secure two-party / multi-party computation [72, 18, 4, 27], with proofs given in the universal composability model [12, 14].

While the execution of the ML-DSA key generation algorithm on top of a 2PC / MPC protocol set is relatively straightforward, if one can create *small* private values (which may be reduced to the conversion of private values between different moduli of sharing [63, 21]), the signature creation is trickier. Besides the generation of small private values (same as key generation), it has two main parts: high bits computation, and rejection sampling. The latter is mostly inequality checking, for which a number of MPC protocols have been proposed [57, 2, 74]; we are adapting [2] for our purposes. The high bits computation mostly consists of division (with public divisor) together with truncation / rounding of the result. Protocols for secure division have been proposed [7, 70, 69], but they are not much related to our techniques. Our division protocol is highly specific to the used divisor.

Our actively secure two-party protocols have subprotocols that provide active security against one of the parties, but only active *privacy* against the other party. This consideration of privacy as a separate goal is inspired by [6, 65]. Similar considerations are present in certain offline precomputation protocols of MPC protocol sets [36], but our methods of composition appear to be novel.

## 2 Preliminaries

### 2.1 Notation

Let $a \xleftarrow{\$} A$ denote a uniformly random sampling of an element $a$ from the set $A$. Symbol $\perp$ is used to indicate a failure or rejection. $\mathbb{Z}$ denotes the set of integers, $\mathbb{N}$ the set of non-negative integers, $\mathbb{Z}_q$ the set of integers modulo $q$. The set $\{0, 1, \ldots, n-1\}$ is denoted by $[n]$. Let $R_q$ denote the ring of polynomials

[13] Some results of this work were presented in [50]

[14] Since Crystals-Dilithium (ML-DSA submission name to NIST PQC competition) was inspired by Star Trek series, we decided to call our protocol Trilithium, a compound from Star Trek Generations

$\mathbb{Z}_q[X]/(X^n+1)$, where $q$ is a prime number and $n \in \mathbb{N}$. We denote scalar values (including polynomials) by italic letters, vectors by bold lowercase latters, and matrices by bold uppercase letters.

For $\alpha \in \mathbb{N}$ and $x \in \mathbb{Z}$, let $x' = x \bmod^{\pm} \alpha$ denote centered modulo reduction, where $x' \equiv x \pmod{\alpha}$ and either $-\frac{\alpha}{2} < x' \leq \frac{\alpha}{2}$ (if $\alpha$ is even) or $-\frac{\alpha-1}{2} \leq x' \leq \frac{\alpha-1}{2}$ (if $\alpha$ is odd).

For an element $x \in \mathbb{Z}_q$, its infinity norm is defined as $\|x\|_\infty = |x \bmod^{\pm} q|$, where $|x|$ denotes the absolute value of the element. For an element $p \in R_q$, its infinity norm is defined as $\|p\|_\infty = \max_i \|p_i\|_\infty$. For a vector $\mathbf{v}$, it is $\|\mathbf{v}\|_\infty = \max_i \|v_i\|_\infty$.

Let $v \in [B]$. The *characteristic vector* of $v$ is a vector $\mathsf{cv}(v) = (t_0, t_1, \ldots, t_{B-1})$, where $t_v = 1$ and $t_i = 0$ for all $i \neq v$. The length of $\mathsf{cv}(v)$, i.e. the upper bound $B$ will be clear from the context.

### 2.2 ML-DSA (Module-Lattice-based Digital Signature Algorithm)

Module-Lattice-Based Digital Signature Algorithm (ML-DSA) is a quantum-safe digital signature scheme. ML-DSA follows Fiat-Shamir with Aborts [56] paradigm and its security relies on hardness of MLWE, MSIS and SelfTargetMSIS problems.

Ducas et al. [35, Fig. 1] present a "template" signature scheme, which is later improved to Crystals-Dilithium and ML-DSA by significantly shortening the public key and adding some *hints* to signatures. The integrity properties of the "template" scheme are no worse than those of ML-DSA. The public key is shortened by simply cutting off some of its bits. Given a public key and a signature of the "template" scheme, one can compute the *hints* for ML-DSA, resulting in a valid ML-DSA signature. Hence it is sufficient to give a two-party protocol for key generation and signing in the "template" scheme; both parties locally store the public key of the "template" scheme resulting from the key generation protocol, publish it without the to-be-cut-off bits, and locally add the hints to the signatures produced by the signing protocol. In the following, let us describe the "template" scheme; see [35] and FIPS 204 [59] for the description of the full scheme.

The parameters of the "template" scheme are a prime number $q$, data sizes $k, \ell, n, \tau$, and value sizes $\gamma_1, \gamma_2, \eta$. Also define $\alpha = 2\gamma_2$ and $\beta = \eta \cdot \tau$. The values must such that $\alpha$ divides $(q-1)$. We use the ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$. Let $S_m \subseteq R_q$ [resp. $\bar{S}_m \subseteq R_q$] for $m \in \mathbb{N}$ denote the set of polynomials with all coefficients between $-m$ [resp. $-m+1$] and $m$. Let $B_\tau \subseteq S_1$ denote the set of polynomials with exactly $\tau$ non-zero coefficients.

Given a value $w \in \mathbb{Z}_q = \{0, \ldots, q-1\}$, we define $\mathsf{HighBits}(w) \in \mathbb{Z}_q$ as 0 if $w' = q-1$, and $\lfloor w'/\alpha \rfloor$ otherwise, where $w' = (w + \alpha/2 - 1) \bmod q$. This definition is equivalent to [35, Fig. 3]. Also equivalent is defining $\mathsf{LowBits}(w) := (w - \alpha \cdot \mathsf{HighBits}(w)) \bmod q$. One commonly uses superscript $(\cdot)^H$ and $(\cdot)^L$ for high and low bits. The definitions of high and low bits are extended from $\mathbb{Z}_q$ to $R_q$ coefficient-wise, and to vectors of polynomials component-wise.

**Output:** Keypair $pk$, $sk$
$\mathbf{A} \xleftarrow{\$} R_q^{k \times \ell}$ ; // Generated from a seed
$(\mathbf{s}_1, \mathbf{s}_2) \xleftarrow{\$} S_\eta^\ell \times S_\eta^k$
$\mathbf{t} \leftarrow \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$
**return** $pk = (\mathbf{A}, \mathbf{t})$, $sk = (\mathbf{s}_1, \mathbf{s}_2)$

**Algorithm 1:** Key generation in "template" scheme

**Input:** Keypair $(\mathbf{A}, \mathbf{t}; \mathbf{s}_1, \mathbf{s}_2)$, message digest $\mu$
**Output:** Signature $\sigma = (\mathbf{z}, c)$, or $\perp$
$\mathbf{y} \xleftarrow{\$} \bar{S}_{\gamma_1}^\ell$ ; // Follows [59], not [35]
$\mathbf{w} \leftarrow \mathbf{A} \cdot \mathbf{y}$
$c \in B_\tau \leftarrow H(\mu, \mathbf{w}^H)$
$\mathbf{z} \leftarrow \mathbf{y} + c \cdot \mathbf{s}_1$
$\mathbf{r}_0 \leftarrow \mathsf{LowBits}(\mathbf{w} - c \cdot \mathbf{s}_2)$
**if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ *and* $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$ **then**
  **return** $(\mathbf{z}, c)$
**else**
  **return** $\perp$

**Algorithm 2:** Signing attempt in "template" scheme: SignAtt

The signature scheme employs "hashing into a ball". It defines a hash function $H$, based on SHAKE [58], whose output is uniformly distributed in $B_\tau$. See [35, 59] for its precise description.

The key generation algorithm of the "simplified" scheme [35, Fig. 1] is given in Alg. 1. The algorithm for a single signing *attempt* is given in Alg. 2. If Alg. 2 returns $\perp$, then the signer is expected to rerun it, with newly selected $\mathbf{y}$. The ML-DSA specification has picked the parameters so, that the average number of reruns is 4 to 5. In this paper, we give a two-party protocol for Alg. 2. This does not reduce the security of our protocols — the frequency of retries is independent of the private key.

Our protocols also make use of the verification algorithm of the template scheme. The verification consists of making sure that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, computing $\mathbf{x}_0 \leftarrow \mathbf{A} \cdot \mathbf{z} - c \cdot \mathbf{t}$, and checking that $c = H(\mu, \mathbf{x}_0^H)$. Note that $\mathbf{x}_0 = \mathbf{w} - c \cdot \mathbf{s}_2$.

Let us recall the values of the most important parameters in the three parameter sets of ML-DSA [59]: $q = 2^{23} - 2^{13} + 1$, $\alpha \in \{190464, 523776\}$, $\eta \in \{2, 4\}$, $\gamma_1 \in \{2^{17}, 2^{19}\}$. We depend on certain properties of these values to make our protocols efficient.

### 2.3 Module Short Integer Solution

While the unforgeability of ML-DSA, and hence also the property of our protocols to constitute a secure threshold signature scheme is based on certain lattice-based hardness assumptions, we also "independently" use one of these assumptions to show that an adversarial party cannot cheat. Let us state it here. Given parameters $q, n, m, \eta \in \mathbb{N}$, the MSIS assumption [52] states that for a uniformly sampled $\mathbf{A} \xleftarrow{\$} R_q^{n \times m}$ it is infeasible to find a *non-zero* vector $\mathbf{x} \in R_q^{m+n}$, such that $[\mathbf{A}|\mathbf{I}] \cdot \mathbf{x} = \mathbf{0} \pmod q$ and $\|\mathbf{x}\|_\infty \leq \eta$.

### 2.4 Secure Multiparty Computation

Our protocol, Trilithium, is an instance of secret-sharing based secure multiparty computation (MPC) protocols, where the intermediate values have been additively shared (often modulo "the ML-DSA modulus" $q$) between two parties: the *Phone* and the *Server*. In these protocols, a private value $v \in \mathbb{Z}_q$ is stored as $[\![v]\!] = ([\![v]\!]_\mathsf{P}, [\![v]\!]_\mathsf{S})$, where $v = [\![v]\!]_\mathsf{P} + [\![v]\!]_\mathsf{S}$, Phone knows the random value $[\![v]\!]_\mathsf{P} \in \mathbb{Z}_q$ and Server knows the value (also random) $[\![v]\!]_\mathsf{S} \in \mathbb{Z}_q$. We may write $[\![v]\!]^{(q)}$ to emphasize that sharing is done modulo $q$.

There exist protocols for arithmetic operations with additively shared values. While linear (including affine) operations can be performed locally by Phone and Server, others require communication between paries. Such non-linear operations may benefit from some *correlated random values (CR)* distributed before the protocol start. For certain operations, e.g. multiplication, there exist protocols for generating CR [45]. Alternatively, some deployments may warrant the use of a trusted party, the *Correlated Randomness Provider (CRP)* to generate and distribute CR. In this paper, we have chosen the CRP approach.

Storing $v$ as $[\![v]\!]$ does not prevent a malicious party from tampering with its value by changing its share. It can be avoided with the help of (homomorphic) "*message authentication codes*" (MACs) [4, 27]. In this paper, we use so-called *BeDOZa-style MACs* [4], where the Phone and the Server have random private MAC keys $\Delta_\mathsf{P}, \Delta_\mathsf{S}$, respectively, and the value $v$ is stored as $\langle\!\langle v \rangle\!\rangle = ([\![v]\!], [\![M_\mathsf{S}]\!], [\![M_\mathsf{P}]\!])$, where $M_\mathsf{S} = [\![v]\!]_\mathsf{P} \cdot \Delta_\mathsf{S}$ and $M_\mathsf{P} = [\![v]\!]_\mathsf{S} \cdot \Delta_\mathsf{P}$. We call $M_\mathsf{S}$ "*Server's MAC on* $[\![v]\!]$" and $M_\mathsf{P}$ "*Phone's MAC on* $[\![v]\!]$". Affine operations with MAC-ed values $\langle\!\langle v \rangle\!\rangle$ are still possible without communication.

Whenever the Phone $\mathsf{P}$ sends its share $[\![v]\!]_\mathsf{P}$ to a Server $\mathsf{S}$, it must accompany it with $[\![M_\mathsf{S}]\!]_\mathsf{P}$. $\mathsf{S}$ is able to verify the correctness of the MAC by checking whether $[\![v]\!]_\mathsf{P} \cdot \Delta_\mathsf{S} = [\![M_\mathsf{S}]\!]_\mathsf{S} + [\![M_\mathsf{S}]\!]_\mathsf{P}$. If $\mathsf{P}$ tries to change $[\![v]\!]_\mathsf{P}$, then its MAC share has to be correspondingly changed, and this task is as hard as guessing the random $\Delta_\mathsf{S}$. The probability of guessing is $1/q$; and if this value is too large for given $q$, then several independent MAC keys $\boldsymbol{\Delta}$ and MACs can be used. Independent MAC keys $\boldsymbol{\Delta}^{(q_i)}$ are also used if the protocol makes use of several different moduli $q_i$. Instead of shares of the MACs, their digests can be communicated, hence MACs do not noticeably contribute to the online complexity of protocols.

While Trilithium two-party protocol (including an honest CRP) is secure against a malicious $\mathsf{P}$ or a malicious $\mathsf{S}$, it includes subprotocols that by their

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that party $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{keygen}, sid)$ from both $\mathsf{P}$ and $\mathsf{S}$, use Alg. 1 to generate a new keypair $(\mathbf{A}, \mathbf{t}; \mathbf{s}_1, \mathbf{s}_2)$. Send $(\mathsf{pkey}, sid, \mathbf{A}, \mathbf{t})$ to $\mathcal{S}$. If neither party is corrupted, then receive $(\mathsf{proceed}, sid)$ from the adversary, otherwise receive $(\mathsf{update}, sid, \mathbf{A}', \mathbf{t}')$ from the adversary. Send $(\mathsf{pkey}, sid, \mathbf{A}, \mathbf{t})$ to the honest party(s) in $\mathcal{Z}$, and $(\mathsf{pkey}, sid, \mathbf{A}', \mathbf{t}')$ to the corrupted party (if any). Accept no further $(\mathsf{keygen})$-commands.
- On input $(\mathsf{sign}, sid, \mu)$ from both the $\mathsf{P}$ and $\mathsf{S}$, compute $\sigma \leftarrow \mathsf{SignAtt}(\mathbf{A}, \mathbf{t}; \mathbf{s}_1, \mathbf{s}_2; \mu)$. Send $(\mathsf{signature}, sid, \mu, \sigma)$ to $\mathcal{S}$. If neither party is corrupted, then receive $(\mathsf{proceed}, sid)$ from the adversary, otherwise receive $(\mathsf{update}, sid, \sigma')$ from the adversary. Send $(\mathsf{signature}, sid, \sigma)$ to the honest party(s) in $\mathcal{Z}$, and $(\mathsf{signature}, sid, \sigma')$ to the the corrupted party (if any).

**Functionality 1:** $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$: ideal functionality for the 2-party ML-DSA

own provide security only against a malicious $\mathsf{P}$ and a semi-honest $\mathsf{S}$, while still providing *privacy* against a malicious $\mathsf{S}$. Here "*privacy*" means that nothing is learned from the *messages* exchanged by the protocol. However, a malicious $\mathsf{S}$ may be able to change the outputs of such protocols, as well as learn something from its public outputs. In the following, we say that such (sub-)protocols are "*AP-secure*", while we call protocols with active security against both parties "*AA-secure*". This is made formal in Sec. 4. In our AP-secure protocols, we often use the representation $\langle\!|v|\!\rangle = (\llbracket v \rrbracket, \llbracket M_{\mathsf{S}} \rrbracket)$ for private values.

Let us introduce notation to explicitly refer to the components of $\langle\langle v \rangle\rangle = (\llbracket v \rrbracket, \llbracket M_{\mathsf{S}} \rrbracket, \llbracket M_{\mathsf{P}} \rrbracket)$, where $M_{\mathsf{S}}$ and $M_{\mathsf{P}}$ are Server's and Phone's MACs on $\llbracket v \rrbracket$. Denote the three components of $\langle\langle v \rangle\rangle$ by ${}_{\mathsf{v}}\langle\langle v \rangle\rangle$, ${}_{\mathsf{ms}}\langle\langle v \rangle\rangle$, and ${}_{\mathsf{mp}}\langle\langle v \rangle\rangle$ respectively. We may also write e.g. ${}_{\mathsf{mp}}\langle\langle v \rangle\rangle_{\mathsf{S}}$ for $\llbracket M_{\mathsf{P}} \rrbracket_{\mathsf{S}}$. The notation ${}_{\mathsf{v}}\langle\!|v|\!\rangle$ and ${}_{\mathsf{ms}}\langle\!|v|\!\rangle$ has similar meaning. We will use this notation in the security proofs, in order to avoid introducing separate notations for the components of secret-shared values.

### 2.5 Universal Composability

We prove the security of our protocols in the *Universal Composability (UC)* framework [12] (our notation is more similar to the functionally equivalent *reactive simulatability* framework [66]). Our real system $\Pi_{\mathsf{ML\text{-}DSA}}$ and ideal functionality $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ (given in
Func. 1) expose an interface to the environment $\mathcal{Z}$ that corresponds to two parties, called $\mathsf{P}$(hone) and $\mathsf{S}$(erver) being able to run a single instance of key generation, and any number of signings. We will construct the protocol $\Pi_{\mathsf{ML\text{-}DSA}}$ and show that it *securely implements* $\mathcal{F}_{\mathit{ML\text{-}DSA}}$, meaning that for each adversary $\mathcal{A}$ against it, there exists an adversary $\mathcal{S}$, such that no environment $\mathcal{Z}$ can distinguish whether it is running in the collection $\mathcal{Z} \| \Pi_{\mathsf{ML\text{-}DSA}} \| \mathcal{A}$ or $\mathcal{Z} \| \mathcal{F}_{\mathsf{ML\text{-}DSA}} \| \mathcal{S}$. We demonstrate the existence of $\mathcal{S}$ by constructing a *simulator Sim* and showing that $\mathcal{S} = \mathit{Sim} \| \mathcal{A}$ is a suitable choice. This is shown by arguing that no $\mathcal{Z} \| \mathcal{A}$ can distinguish $\Pi_{\mathsf{ML\text{-}DSA}}$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}} \| \mathit{Sim}$.

In our proof, we make heavy use of composability: if an ideal functionality $\mathcal{F}'$ is a component in the protocol $\Pi$ that securely implements the ideal functionality $\mathcal{F}$, and if $\Pi'$ is a secure implementation of $\mathcal{F}'$, then $\Pi$, where $\mathcal{F}'$ is replaced with $\Pi'$, also securely implements $\mathcal{F}$. We say that $\Pi$ securely implements $\mathcal{F}$ *in* $\mathcal{F}'$*-hybrid model*. In particular, the CRP will be modeled as a set of ideal functionalities.

A protocol $\Pi$ is hence implemented by a system consisting of several executing components. In our setting, this system always contains two machines, $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$, executing the protocol logic on behalf of $\mathsf{P}$ and $\mathsf{S}$. These machines expose an interface to the environment $\mathcal{Z}$. Besides these two machines, the system may also contain a number of *ideal components* $\mathcal{F}_i$. These components may correspond to subprotocols, or to the generation of CR. In practice, the former are replaced with their secure implementations, while the latter are collected to another party — the CRP.

The machines $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$ take commands from the environment, send messages to each other, issue commands to the ideal components in $\Pi$, receive answers from them, and give answers back to the environment. One of $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$ may receive the corrupt-command from the adversary at the beginning of the execution. In this case, this machine $\mathcal{M}$ hands all control over to the adversary: it sends to the adversary all commands / messages / answers it receives from the environment, the other machine or the ideal components, and it will send out only the answers / messages / commands as told by the adversary. Note that the latter includes the answer that $\mathcal{M}$ sends to the environment.

## 3 Trilithium: Server-Supported ML-DSA Signature Scheme

In this section, we define Trilithium and describe our subprotocols, using the notation typical for MPC protocols. Our descriptions are bottom-up, starting with small operations and finishing with the protocols for key generation and signing. We give informal correctness and security arguments together with the descriptions; formal security arguments are given in the next section.

In our protocols, we make use of the protocol randbit() that takes no inputs and returns $\langle\!\langle b \rangle\!\rangle^{(q)}$ for a random $b \in \{0, 1\}$ and a modulus $q$ that will be clear from the context. We use the bit conversion protocol convbit($\langle\!\langle b \rangle\!\rangle^{(2)}$) that takes a bit $b$ shared modulo 2, and returns $\langle\!\langle b \rangle\!\rangle^{(q)}$ — the same bit shared modulo $q$. Obviously, we also make use of protocols for adding and multiplying shared values, and for generating public random values modulo some prime number $Q$ (called pubrand()). All these protocols are standard, and described e.g. in [73]. We use the CRP for any correlated randomness that these protocols require.

### 3.1 Declassification

Let us start with the *declassification* protocol declassify given in Alg. 3. As discussed in Sec. 2.4, it involves the checking of MACs, using the MAC keys $\boldsymbol{\Delta}_\mathsf{P}^{(q)}$

**Input:** $\langle\langle v \rangle\rangle = (\llbracket v \rrbracket, \llbracket M_{\mathsf{S}} \rrbracket, \llbracket M_{\mathsf{P}} \rrbracket)$, where $v \in \mathbb{Z}_q$, $M_{\mathsf{S}} = \llbracket v \rrbracket_{\mathsf{P}} \cdot \boldsymbol{\Delta}_{\mathsf{S}}$ and $M_{\mathsf{P}} = \llbracket v \rrbracket_{\mathsf{S}} \cdot \boldsymbol{\Delta}_{\mathsf{P}}$
**Output:** $v$
1 $\mathsf{S} \to \mathsf{P}$: $\llbracket v \rrbracket_{\mathsf{S}}$ and $\llbracket M_{\mathsf{P}} \rrbracket_{\mathsf{S}}$
2 $\mathsf{P}$: **stop** if $\llbracket v \rrbracket_{\mathsf{S}} \cdot \boldsymbol{\Delta}_{\mathsf{P}} - \llbracket M_{\mathsf{P}} \rrbracket_{\mathsf{P}} \neq \llbracket M_{\mathsf{P}} \rrbracket_{\mathsf{S}}$
3 $\mathsf{P} \to \mathsf{S}$: $\llbracket v \rrbracket_{\mathsf{P}}$ and $\llbracket M_{\mathsf{S}} \rrbracket_{\mathsf{P}}$
4 $\mathsf{S}$: **stop** if $\llbracket v \rrbracket_{\mathsf{P}} \cdot \boldsymbol{\Delta}_{\mathsf{S}} - \llbracket M_{\mathsf{S}} \rrbracket_{\mathsf{S}} \neq \llbracket M_{\mathsf{S}} \rrbracket_{\mathsf{P}}$
5 $\mathsf{P}$ and $\mathsf{S}$: **return** $\llbracket v \rrbracket_{\mathsf{P}} + \llbracket v \rrbracket_{\mathsf{S}}$

**Algorithm 3:** Declassification protocol: declassify

**Par:** $Q$ — a small prime number
**Output:** $\langle\langle v \rangle\rangle^{(q)}$, $v \xleftarrow{\$} \{0, \ldots, Q-1\}$
**CR:** $\langle\langle \mathbf{t} \rangle\rangle^{(q)}$, s.t. $|\mathbf{t}| = Q$, $\mathbf{t}$ is a one-hot 0-1 vector
1 $r \leftarrow \mathsf{pubrand}() \in \mathbb{Z}_Q$
2 **return** $\sum_{i=0}^{Q-1} i \cdot \langle\langle t_{(i+r) \bmod Q} \rangle\rangle$

**Algorithm 4:** Generation of a small random value: $\Pi_{\mathsf{gen\_small}}$

and $\boldsymbol{\Delta}_{\mathsf{S}}^{(q)}$ for the modulus $q$. The inputs to the declassification protocol are the shares $\llbracket v \rrbracket$ of the value $v$, and the shares of Server's MAC $M_{\mathsf{S}}$ and Phone's MAC $M_{\mathsf{P}}$ for this sharing $\llbracket v \rrbracket$.

For a small modulus $q$, the shares of MACs would make up the lion's share of the communication. In our actual implementations, the shares of MACs are actually not sent. Instead, the hash of all MAC shares to be sent in the current round is computed, and only this hash value is sent to the other party. The other party can compute the same hash from the values it holds, and verify that it equals the one it received.

We see that Alg. 3 is asymmetric: $\mathsf{P}$ learns the value $v$ first. This will later be reflected in our security proofs.

Alg. 3 is also a good description of the declassification of $\langle\!|v|\!\rangle$, if we leave out the checks related to $\mathsf{P}$'s MAC. We get the AP-declassification by removing $M_{\mathsf{P}}$ and line 2 from Alg. 3.

### 3.2 Randomness generation

The protocol gen_small, parametrized by a small prime number $Q$ and given in Alg. 4, takes no inputs and returns a uniform random value $\langle\langle v \rangle\rangle^{(q)}$, such that $0 \leq v < Q$. The correlated randomness used by this algorithm is a private 0-1 vector, where the position of the only value "1" is randomly selected by the CRP, and unknown to $\mathsf{P}$ and $\mathsf{S}$.

In the rest of Sec. 3 we distinguish between parameters (fixed at compile-time) and inputs (known only at run-time) to a protocol. We write $f[c](a)$ to denote the invocation of a protocol $f$ with parameter $c$ and input $a$. We may omit the parameters if they're clear from the context. The correlated randomness **CR** used by the protocols is assumed to be uniformly distributed, subject to the listed constraints.

**Par:** ML-DSA parameters $k$, $\ell$, $\eta$
**Input:** The matrix $\mathbf{A}$ of the public key
**Output:** The private key $\langle\langle \mathbf{s}_1 \rangle\rangle$, $\langle\langle \mathbf{s}_2 \rangle\rangle$
**foreach** $i \in [k+\ell]$, $t \in [256]$ **do**
  **if** $\eta = 2$ **then**
    $\langle\langle r_{i,t} \rangle\rangle \leftarrow \mathsf{gen\_small}[5]() - 2$
  **else** // $\eta = 4$
    $\langle\langle r_{i,t} \rangle\rangle \leftarrow 3 \cdot \mathsf{gen\_small}[3]() + \mathsf{gen\_small}[3]() - 4$
$\langle\langle \mathbf{s}_1 \rangle\rangle \leftarrow (\sum_{t=0}^{255} r_{i-1,t} X^t)_{i=1}^{\ell}$
$\langle\langle \mathbf{s}_2 \rangle\rangle \leftarrow (\sum_{t=0}^{255} r_{i+\ell-1,t} X^t)_{i=1}^{k}$
**return** $\langle\langle \mathbf{s}_1 \rangle\rangle$, $\langle\langle \mathbf{s}_2 \rangle\rangle$

**Algorithm 5:** ML-DSA private key generation: keygen

### 3.3 Private key generation

Generation of private small random values is all that is needed for ML-DSA key generation. The protocol keygen is given in Alg. 5. It generates the necessary number of random values and arranges them as coefficients of polynomials (with $X$ denoting the formal variable). Here and in the following, we do not make explicit whether polynomials are represented coefficient-wise or in the Number Theoretic Transform (NTT) representation. Note that the conversion between the two is a linear operation. Our implementation converts whereever reasonable. In particular, the multiplications of polynomials are done using NTT.

It would perhaps be natural to have Alg. 5 also contain public key generation, starting from the generation of a random matrix $\mathbf{A}$, and finishing with the (linear) computation of $\langle\langle \mathbf{t} \rangle\rangle \leftarrow \mathbf{A} \cdot \langle\langle \mathbf{s}_1 \rangle\rangle + \langle\langle \mathbf{s}_2 \rangle\rangle$ and its declassification. For the uniformity of our security arguments (Sec. 4), these steps are not included here. But they are part of $\Pi_{\mathsf{ML\text{-}DSA}}$.

### 3.4 Zero equality checking protocol

A necessary sub-protocol in distributed signing is the equality check for values shared modulo $q$. This is equivalent to checking whether a value is 0: the input to zero_check, given in Alg. 6 is a value $\langle\langle v \rangle\rangle$, and the output is $\langle\langle b \rangle\rangle$, where $b = 1$ iff $v = 0$. While protocols for this task have been proposed before [23], our protocol is simplified by the guarantee $v < B$ for a small upper bound $B$ parametrizing the zero_check protocol. With this guarantee, we can keep the size of the correlated randomness proportional to $B$, not to the much larger modulus $q$.

Let us show that if $v < B$ then Alg. 6 indeed produces the correct result. We have denoted $a = \lfloor q/B \rfloor$. The vector $\mathbf{t}$ is constructed so that the bit $t_{\lfloor m/a \rfloor}$ is equal to 1, while all other bits are equal to 0. There are two cases.
**Case 1:** $v = 0$. Then $d = m$. The protocol outputs the secret-shared bit $t_{\lfloor m/a \rfloor}$. By the construction of the vector $\mathbf{t}$, this bit is equal to 1.
**Case 2:** $v \neq 0$. The protocol outputs the secret-shared bit $t_{\lfloor d/a \rfloor}$. If we consider all values as integers (i.e. elements of $\mathbb{Z}$, not elements of $\mathbb{Z}_q$), then we can write

**Par:** $B \in \mathbb{N}$. Denote $a = \lfloor q/B \rfloor$
**Input:** $\langle\langle v \rangle\rangle^{(q)}$, where $v < B$
**Output:** $\langle\langle b \rangle\rangle^{(q)}$, $b \in \{0, 1\}$, $b = 1 \Leftrightarrow v = 0$
**CR:** $\langle\langle m \rangle\rangle^{(q)}$. $\langle\langle \mathbf{t} \rangle\rangle^{(q)}$, s.t. $|\mathbf{t}| = B + 1$, $\mathbf{t} = \mathsf{cv}(\lfloor m/a \rfloor)$
1 $d \leftarrow \mathsf{declassify}(\langle\langle m \rangle\rangle + a \cdot \langle\langle v \rangle\rangle)$
2 **return** $\langle\langle t_{\lfloor d/a \rfloor} \rangle\rangle$

**Algorithm 6:** Checking whether a value is 0: zero_check

$\lfloor \frac{d}{a} \rfloor = \lfloor \frac{(m+av) \bmod q}{a} \rfloor$. We have $m < q$ and $av \leq \frac{q}{B} \cdot v < \frac{q}{B} \cdot B = q$. Hence $0 \leq m + av < 2q$. Let us consider two cases.
**Case 2a:** $m + av < q$. Then $\lfloor \frac{d}{a} \rfloor = \lfloor \frac{m+av}{a} \rfloor = \lfloor \frac{m}{a} \rfloor + v \neq \lfloor \frac{m}{a} \rfloor$. Hence by the construction of the vector $\mathbf{t}$, the bit $t_{\lfloor d/a \rfloor}$ is equal to 0.
**Case 2b:** $q \leq m + av < 2q$. Then $d = m + av - q \leq m + \frac{q}{B} \cdot (B - 1) - q \leq m + (q - a) - q = m - a$. Hence $\lfloor \frac{d}{a} \rfloor \leq \lfloor \frac{m-a}{a} \rfloor = \lfloor \frac{m}{a} \rfloor - 1$. We obtain $\lfloor d/a \rfloor \neq \lfloor m/a \rfloor$, hence the bit $t_{\lfloor d/a \rfloor}$ is equal to 0.

### 3.5 Symmetric reshare to parts

In order to compute the *high bits* of a value $w \in \mathbb{Z}_q$, we have to decompose it into high and low bits. Consider a vector $\mathbf{r} = (r_0, r_1, \ldots, r_{n-1}) \in \mathbb{N}^n$, such that $R_n \geq 2q$, where we denote $R_0 = 1$ and $R_i = R_{i-1} \cdot r_{i-1}$. Given a value $v \in [2q]$, there exist unique $v_0, v_1, \ldots, v_{n-1}$, such that $0 \leq v_i < r_i$ and $v = \sum_{i=0}^{n-1} R_i \cdot v_i$. We call $(v_0, \ldots, v_{n-1})$ the *representation of $v$ in the mixed-radix base* $\mathbf{r}$, and call the individual values $v_i$ the *digits* of $v$ (wrt. $\mathbf{r}$).

Our protocol symm_rp, called "*symmetric reshare to parts*" and parametrized by a mixed-radix basis $\mathbf{r}$, takes as input a shared value $\langle\langle w \rangle\rangle$ and returns the secret-shared digits of $w$ in the representation of their characteristic vectors (CV) (where the length of the CV of the $i$-th digit is $r_i$). Actually, the digits may correspond either to $w$ or to $w + q$; this is chosen randomly by symm_rp, given in Alg. 7.

We see that Alg. 7 begins by making public the value $y = (w - s) \bmod q$. The loop that follows basically adds the values $y$ and $s$, thus explaining why the result randomly corresponds to either $w$ or $w + q$ (but never to $2q - 1$). This addition is digit-wise. To add a digit with the CV $\mathbf{d}_i$, and the digit $z_i$, we rotate the vector to the right by $z_i$ positions, thereby increasing the index of the position with "1" on it by $z_i$. As the length of $\mathbf{d}_i$ is $2r_i$, this rotation (also with $z_i + 1$) cannot move the element 1 of the vector over its right end. The rotation with $z_i$ corresponds to no carry coming into the $i$-th digit, while the rotation with $z_i + 1$ corresponds to the incoming carry. The actual carry bit, $c_i$, is used to *obliviously choose* between the two possibilities. The oblivious choice operation, $b \,?\, x : y$ is defined to be $x$ if $b = 1$, and $y$ if $b = 0$. It can be computed as $y + b \cdot (x - y)$. In Alg. 7 we use it for vector-valued $x$ and $y$; in our implementation, the CR we use is for scalar-vector multiplication.

During this addition, we have an outgoing carry at the $i$-th digit if that digit is at least $r_i$, hence we find that carry by summing up the indicator bits of the

**Par:** $\mathbf{r} = (r_0, \ldots, r_{n-1})$, where $\prod r_i \geq 2q$
**Input:** $\langle\langle w \rangle\rangle^{(q)}$
**Output:** $\langle\langle \mathbf{b}_0 \rangle\rangle^{(q)}, \ldots, \langle\langle \mathbf{b}_{n-1} \rangle\rangle^{(q)}$, CV-s of digits of either $w$ or $w + q$ (but never of $2q - 1$)
**CR:** $\langle\langle s \rangle\rangle^{(q)}$. $\langle\langle \mathbf{d}_0 \rangle\rangle^{(2)}, \ldots, \langle\langle \mathbf{d}_{n-1} \rangle\rangle^{(2)}$: CV-s of digits of $s$

$\forall i$: append $\langle\langle \mathbf{d}_i \rangle\rangle$ with $\langle\langle 0 \rangle\rangle$-s, s.t. its length becomes $2r_i$
$y \leftarrow \mathsf{declassify}(\langle\langle w \rangle\rangle - \langle\langle s \rangle\rangle)$
$(z_0, \ldots, z_{n-1}) \leftarrow$ the digits of $y$
**for** $i = 0$ **to** $n - 1$ **do**
  $\langle\langle \mathbf{p}'_i \rangle\rangle \leftarrow \mathsf{rotate\_right}(\langle\langle \mathbf{d}_i \rangle\rangle, z_i)$
  $\langle\langle \mathbf{p}''_i \rangle\rangle \leftarrow \mathsf{rotate\_right}(\langle\langle \mathbf{d}_i \rangle\rangle, z_i + 1)$
  $\langle\langle \mathbf{p}_i \rangle\rangle \leftarrow$ **if** $i = 0$ **then** $\langle\langle \mathbf{p}'_0 \rangle\rangle$ **else** $\langle\langle c_i \rangle\rangle$ ? $\langle\langle \mathbf{p}''_i \rangle\rangle : \langle\langle \mathbf{p}'_i \rangle\rangle$;
  $\langle\langle c_{i+1} \rangle\rangle \leftarrow \sum_{j=r_i}^{2r_i-1} \langle\langle p_{i,j} \rangle\rangle$
**foreach** $i \in [n]$, $j \in [r_i]$ **do**
  $\langle\langle b_{i,j} \rangle\rangle^{(q)} \leftarrow \mathsf{convbit}(\langle\langle p_{i,j} \rangle\rangle^{(2)} + \langle\langle p_{i,j+r_i} \rangle\rangle^{(2)})$
**return** $\langle\langle \mathbf{b}_0 \rangle\rangle, \ldots, \langle\langle \mathbf{b}_{n-1} \rangle\rangle$

**Algorithm 7:** Symmetric reshare to parts: symm_rp

upper half of the vector $\mathbf{p}_i$. The final value of that digit comes either from the lower (if there was no carry) or the upper (if there was carry) half of the vector $\mathbf{p}_i$.

In our implementation, we first propagate all carries as in carry-lookahead adders [32], and only afterwards compute the vectors $\mathbf{p}_i$. We use binomial tuples [67] to compute the carries in a single round of communication. In this way, the main loop only requires two communication rounds, instead of $n - 1$. The binomial tuples will not become too large, because we use $n = 5$.

### 3.6 Distributed HighBits decomposition

The computation of the high bits of a secret-shared value is performed by the high_bits protocol, given in Alg. 8. The algorithm begins by shifting $w$ by $\alpha/2-1$. Indeed, according to the definition of high bits, we have $w^H = \lfloor w'/\alpha \rfloor$, unless $w' = q - 1$, in which case $w^H = 0$. We continue by splitting $w'$ (or $w' + q$) into five digits, let us call these digits $d_0, \ldots, d_4$, their characteristic vectors are $\mathbf{b}_0, \ldots, \mathbf{b}_4$.

Note that $r_0 \cdots r_3 = \alpha$. Hence, if the digits correspond to $w'$, then $\lfloor w'/\alpha \rfloor$ is "basically" $d_4$. If the digits correspond to $w' + q$, then we have added the digits of $q$ to the digits of $w'$ to obtain $d_0, \ldots, d_4$; the digits of $q$ are $(1, 0, 0, 0, s)$. To handle both cases, we should return $d_4$ if $d_4 < s$, and return $d_4 - s$ otherwise. This can be computed from $\mathbf{b}_4$ by finding its inner product with the constant vector $(0, 1, \ldots, s - 1, 0, 1, 2, \ldots)$.

There are exceptions, though. One of them is $w' = q - 1$, in which case the digits are $(0, 0, 0, 0, s)$. We see that this is actually not an exception; we would return $d_4 - s = 0$ as we should. A true exception is that if the digits happen to be $(0, 0, 0, 0, s')$, where $s' > s$, then we should not return $s' - s$, but $s' - s - 1$. We

**Par:** ML-DSA parameter $\alpha = 2 \cdot \gamma_2$. Let $s = (q-1)/\alpha$

Let $\mathbf{r} = (r_0, \ldots, r_4) = \begin{cases} (31, 24, 16, 16, 2s+1), & \text{if } \alpha = 190464 \\ (31, 33, 32, 16, 2s+1), & \text{if } \alpha = 523776 \end{cases}$

**Input:** $\langle\!\langle w \rangle\!\rangle$

**Output:** $\langle\!\langle v \rangle\!\rangle$, s.t. $v = w^H$

$\langle\!\langle w' \rangle\!\rangle \leftarrow \langle\!\langle w \rangle\!\rangle + (\alpha/2 - 1)$

$(\langle\!\langle \mathbf{b}_0 \rangle\!\rangle, \ldots, \langle\!\langle \mathbf{b}_4 \rangle\!\rangle) \leftarrow \mathsf{symm\_rp}[\mathbf{r}](\langle\!\langle w' \rangle\!\rangle)$

$\langle\!\langle f \rangle\!\rangle \leftarrow \sum_{i=0}^{3} \langle\!\langle b_{i,0} \rangle\!\rangle + \sum_{i=s+1}^{2s} \langle\!\langle b_{4,i} \rangle\!\rangle$

$\langle\!\langle c \rangle\!\rangle \leftarrow \mathsf{zero\_check}[6](5 - \langle\!\langle f \rangle\!\rangle)$

**return** $\left(\sum_{i=0}^{s-1} i \cdot \langle\!\langle b_{4,i} \rangle\!\rangle\right) + \left(\sum_{i=s}^{2s} (i-s) \cdot \langle\!\langle b_{4,i} \rangle\!\rangle\right) - \langle\!\langle c \rangle\!\rangle$

**Algorithm 8:** Finding the high bits of a value: $\mathsf{high\_bits}$

**Par:** ML-DSA parameters $k, \ell, \alpha, \gamma_1$

**Input:** $\mathbf{A}$

**foreach** $i \in [\ell]$, $t \in [256]$, $u \in [(\log \gamma_1) + 1]$ **do**

  $\langle\!\langle b_{i,t,u} \rangle\!\rangle \leftarrow \mathsf{randbit}()$

$\langle\!\langle \mathbf{y} \rangle\!\rangle \leftarrow \left(\sum_{t=0}^{255} (-\gamma_1 + \sum_{u=0}^{\log \gamma_1} 2^u \cdot \langle\!\langle b_{i-1,t,u} \rangle\!\rangle) \cdot X^t\right)_{i=1}^{\ell}$

$\langle\!\langle \mathbf{w} \rangle\!\rangle = \left(\sum_{t=0}^{255} \langle\!\langle w_{i-1,t} \rangle\!\rangle \cdot X^t\right)_{i=1}^{k} \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle$

**foreach** $i \in [k]$, $t \in [256]$ **do**

  $\langle\!\langle w^H_{i,t} \rangle\!\rangle \leftarrow \mathsf{high\_bits}(\langle\!\langle w_{i,t} \rangle\!\rangle)$

**return** $\langle\!\langle \mathbf{y} \rangle\!\rangle$, $\langle\!\langle \mathbf{w}^H \rangle\!\rangle$

**Algorithm 9:** Computing the FSwA commitment: $\mathsf{gcom}$

check this by adding up the bits indicating this situation into $f$, and checking that all these bits are 1 (i.e. $5 - f$ is zero).

We point out that the feasibility of Alg. 8 very much relies on the *smoothness* of $\alpha$: we have managed to represent $\alpha$ as the product of a few numbers, none of which are too big to have a vector of private values of this length. This smoothness is not implied by the requirements of ML-DSA: while $q-1$ is required to be divisible by a large power of 2 to enable NTT, the construction of ML-DSA puts no further requirements on the odd prime factors of $q-1$.

With the introduced algorithms, the computation of the commitment $\mathbf{w}^H$ and the secrets corresponding to it is straightforward. We record it in Alg. 9. NTT is used for the multiplication of polynomials. The vector of polynomials $\langle\!\langle \mathbf{w}^H \rangle\!\rangle$ will be declassified outside Alg. 9, the declassification is not included here for the same reason as in Alg. 5.

### 3.7 Arguments to rejection sampling

With $\mathbf{w}^H$, we can compute the challenge $c$ and use it to compute the response $\langle\!\langle \mathbf{z} \rangle\!\rangle = \langle\!\langle \mathbf{y} \rangle\!\rangle + c \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle$. Before releasing $\langle\!\langle \mathbf{z} \rangle\!\rangle$, we have to perform the rejection check on it. We also have to perform the rejection check on $\langle\!\langle \mathbf{r}_0 \rangle\!\rangle = \mathsf{LowBits}(\langle\!\langle \mathbf{w} \rangle\!\rangle - c \cdot \langle\!\langle \mathbf{s}_2 \rangle\!\rangle)$. A rejection check makes sure that the value belongs to a segment. The value and the segment can be shifted, such that the segment starts from 0, hence

the rejection check is an inequality check against a constant, for which we have chosen to adapt the efficient two-party protocol (with CRP) by Attrapadung et al. [2]. Their protocol works for rings $\mathbb{Z}_{2^n}$, and has only passive security. We have managed to adapt it to rings $\mathbb{Z}_q$, and to AP-security. This turns out to be sufficient for an AA-secure ML-DSA protocol, as we explain below.

The computation of $\langle\langle \mathbf{r}_0 \rangle\rangle$ may be complicated due to the need to find the low bits. Fortunately, it turns out that $\mathbf{r}_0$ passes the rejection check iff $\mathbf{x} = \mathsf{LowBits}(\mathbf{w}) - c \cdot \mathbf{s}_2$ passes it. The sharing $\langle\langle \mathbf{x} \rangle\rangle$ is easy to compute, because $\langle\langle \mathbf{w}^L \rangle\rangle = \mathsf{LowBits}(\langle\langle \mathbf{w} \rangle\rangle)$ is a linear combination of $\langle\langle \mathbf{w} \rangle\rangle$ and $\langle\langle \mathbf{w}^H \rangle\rangle$.

Let us show that $\mathbf{r}_0 \leftarrow \mathsf{LowBits}(\mathbf{w} - c \cdot \mathbf{s}_2)$ passes the rejection check iff $\mathbf{x} = \mathsf{LowBits}(\mathbf{w}) - c \cdot \mathbf{s}_2$ passes it with the same upper bound $\gamma_2 - \beta$. By the definition of $\mathsf{LowBits}$, a coefficient of an element of $\mathbf{r}_0$ is between $-\gamma_2$ and $\gamma_2$. The rejection check passes if all coefficients are actually between $-\gamma_2 + \beta + 1$ and $\gamma_2 - \beta - 1$.

A coefficient of an element of $\mathbf{x}$ may be outside the range $[-\gamma_2, \gamma_2]$; it is between $-\gamma_2 - \beta$ and $\gamma_2 + \beta$. In order to transform $\mathbf{x}$ to $\mathbf{r}_0$, we have to add or subtract $2\gamma_2$ to all coefficients that are outside the range $[-\gamma_2, \gamma_2]$. If some coefficient in $\mathbf{x}$ is currently outside $[-\gamma_2, \gamma_2]$, then the addition or subtraction will turn it to a value that will be rejected by the rejection check. Hence the result of rejection-checking $\mathbf{r}_0$ will be the same as rejection-checking $\mathbf{x}$.

### 3.8 Splitting shares to digits

An important step in [2] is locally splitting a share $[\![v]\!]_\mathsf{P}$ or $[\![v]\!]_\mathsf{S}$ into smaller digits, and then thinking of these smaller digits as shares of (smaller) values and working with them. For passive security and rings $\mathbb{Z}_{2^n}$, this splitting is basically a non-operation (as long as the basis elements are powers of 2, too), taking certain bits from the shares. With active security and MACs on the shares, we need the digits of the shares to obtain MACs, too. We cannot do the splitting so, that the digits of both Phone's and Server's share get the MACs, but we can give MACs to digits of one party's share. This is done by the protocol $\mathsf{asymm\_rp}$, given in Alg. 10. Similarly to Alg. 7, it takes a radix basis as a parameter. To simplify this, we let all radixes be equal to $r$. Let $\mathsf{split}[r, d](v)$ return the length-$d$ vector of digits of the value $v$. Its inverse is $\mathsf{split}^{-1}[r, d](\mathbf{v}) = \sum_{i=0}^{d-1} r^i \cdot v_i$.

The input of the Alg. 10 is a value $\langle\!|v|\!\rangle = ([\![v]\!], [\![[\![v]\!]_\mathsf{P} \cdot \boldsymbol{\Delta}_\mathsf{S}]\!])$, where we have denoted the second component with $[\![V]\!]$. The correlated randomness contains a random $m$ known only to $\mathsf{P}$, and Server's MACs on both $m$ and $\mathsf{split}(m)$, denoted $M$ and $\mathbf{r}$, respectively. In the end result, $\mathsf{split}(m)$ will be Phone's share, while Server adjusts its share (before splitting it) by the difference of $[\![v]\!]_\mathsf{P}$ and $m$.

The input and output modulus of Alg. 10 do not have to be the same. In fact, we need the outputs to be shared over a small modulus, in order to input them to the protocols that are analogues to [2]. In Alg. 10, we make explicit, over which modulus each value has been shared. Recall that for different moduli, the Server has different MAC keys $\boldsymbol{\Delta}_\mathsf{S}$. Also, the radix $r$ has to be small enough for the additions in $[\![\mathbf{p}]\!]_\mathsf{P} + [\![\mathbf{p}]\!]_\mathsf{S}$ to not overflow. In our implementation, we use $Q = 29$, $r = 15$ and $d = 6$.

**Par:** output modulus $Q$, radix $r \leq \frac{Q+1}{2}$, output length $d$
**Input:** $\langle\!|v|\!\rangle^{(q)} = (\llbracket v \rrbracket^{(q)}, \llbracket V \rrbracket^{(q)})$
**Output:** $\langle\!|\mathbf{p}|\!\rangle^{(Q)}$, s.t. $\mathsf{split}^{-1}(\llbracket \mathbf{p} \rrbracket_{\mathsf{P}}^{(Q)} + \llbracket \mathbf{p} \rrbracket_{\mathsf{S}}^{(Q)}) = v \pmod q$
**CR:** $m \in \mathbb{Z}_q$ to P. $\llbracket M \rrbracket^{(q)} = \llbracket m \cdot \boldsymbol{\Delta}_{\mathsf{S}}^{(q)} \rrbracket^{(q)}$ and $\llbracket \mathbf{r} \rrbracket^{(Q)} = \llbracket \mathsf{split}(m) \cdot \boldsymbol{\Delta}_{\mathsf{S}}^{(Q)} \rrbracket^{(Q)}$

1 P: $w \leftarrow \llbracket v \rrbracket_{\mathsf{P}}^{(q)} - m$ and $M_w \leftarrow \llbracket V \rrbracket_{\mathsf{P}}^{(q)} - \llbracket M \rrbracket_{\mathsf{P}}^{(q)}$
2 P→S: $w, M_w$
3 S: **stop** if $M_w \neq w \cdot \boldsymbol{\Delta}_{\mathsf{S}}^{(q)} - (\llbracket V \rrbracket_{\mathsf{S}}^{(q)} - \llbracket M \rrbracket_{\mathsf{S}}^{(q)})$
4 S: $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{S}}^{(Q)} \leftarrow (\mathsf{split}((\llbracket v \rrbracket_{\mathsf{S}}^{(q)} + w) \bmod q), \llbracket \mathbf{r} \rrbracket_{\mathsf{S}}^{(Q)})$
5 P: $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}^{(Q)} \leftarrow (\mathsf{split}(m), \llbracket \mathbf{r} \rrbracket_{\mathsf{P}}^{(Q)})$
6 **return** $\langle\!|\mathbf{p}|\!\rangle^{(Q)}$

**Algorithm 10:** Asymmetric reshare to parts: asymm_rp

**Par:** Input modulus $Q$, output modulus $N$
**Input:** $\langle\!|v|\!\rangle^{(Q)}$
**Output:** $\langle\!|\mathbf{b}|\!\rangle^{(N)}$, s.t. $\mathbf{b}$ is the CV of $v$
**CR:** $\langle\!|r|\!\rangle^{(Q)}$ and $\langle\!|\mathbf{d}|\!\rangle^{(N)}$, s.t. $\mathbf{d}$ is the CV of $r$

1 $f \leftarrow \mathsf{declassify}(\langle\!|v|\!\rangle^{(Q)} - \langle\!|r|\!\rangle^{(Q)})$
2 $\forall i \in [Q] : \langle\!|b_i|\!\rangle^{(N)} \leftarrow \langle\!|d_{(i-f) \bmod Q}|\!\rangle^{(N)}$
3 **return** $\langle\!|\mathbf{b}|\!\rangle^{(N)}$

**Algorithm 11:** Characteristic vector: ch_vec

**Par:** Input modulus $Q$, radix $r \leq \frac{Q+1}{2}$, output modulus $N$
**Input:** $\langle\!|v|\!\rangle^{(Q)}$
**Output:** $\langle\!|b|\!\rangle^{(N)}$, $\langle\!|c|\!\rangle^{(N)}$, indicating whether $v = r - 1$ and $v \geq r$, respectively

1 $\langle\!|\mathbf{w}|\!\rangle \leftarrow \mathsf{ch\_vec}[Q, N](\langle\!|v|\!\rangle)$
2 **return** $\langle\!|w_{r-1}|\!\rangle$ *and* $\sum_{i=r}^{Q-1} \langle\!|w_i|\!\rangle$

**Algorithm 12:** Short overflow: short_of

Having split a private value into digits $\langle\!|\mathbf{p}|\!\rangle^{(Q)}$, Attrapadung et al. [2] proceed to convert these digits to characteristic vectors. The protocol for this conversion is straightforward; it is given in Alg 11. The input and output modulus of this protocol may be different, too.

### 3.9 Overflow

The inequality protocol of Attrapadung et al. [2] computes the *overflows* of the to-be-compared private values. Given a vector $\mathbf{p} = (p_0, \ldots, p_{d-1})$, where $p_i \leq 2(r-1)$ (such vector could privately be the output of Alg 10), we say that $\mathbf{p}$ *overflows* if $\sum_{i=0}^{d-1} r^i \cdot p_i \geq r^d$. The overflowing of $\langle\!|\mathbf{p}|\!\rangle$ is computed in two steps, first for a single digit (short_of, Alg. 12) and then for the entire vector (long_of, Alg. 13).

Attrapadung et al. [2] have given an explanation on how Alg. 13 produces the correct output, let us repeat this explanation here. The input to the long overflow protocol (Alg. 13) is a multi-digit number $\mathbf{p}$, where the least significant digit is

**Par:** Length $d$; input, "middle", and output moduli $Q, N, M$ satisfying $N > 2^d$; radix $r \leq (Q+1)/2$
**Input:** $\langle\!|\mathbf{p}|\!\rangle^{(Q)}$ of length $d$
**Output:** Bit $\langle\!|z|\!\rangle^{(M)}$, indicating whether $\mathbf{p}$ overflows

**foreach** $i \in [d]$ **do**
  $(\langle\!|b_i|\!\rangle^{(N)}, \langle\!|c_i|\!\rangle^{(N)}) \leftarrow \mathsf{short\_of}[Q, r, N](\langle\!|p_i|\!\rangle)$
$\langle\!|m|\!\rangle^{(N)} \leftarrow \sum_{i=1}^{d-1} 2^{i-1} \cdot \langle\!|b_i|\!\rangle + \sum_{i=0}^{d-1} 2^i \cdot \langle\!|c_i|\!\rangle$
$\langle\!|\mathbf{k}|\!\rangle^{(M)} \leftarrow \mathsf{ch\_vec}[N, M](\langle\!|m|\!\rangle)$
**return** $\sum_{i=2^{d-1}}^{N-1} \langle\!|k_i|\!\rangle$

**Algorithm 13:** Long overflow: long_of

indexed as 0 and the most significant digit as $d-1$. With the help of the short overflow protocol, we find for each digit $p_i$ whether it overflows (indicated by the bit $c_i$) or "almost overflows" (indicated by the bit $b_i$). Here "almost overflowing" means that the overflowing is propagated: if the next lowest digit overflows then this one will overflow as well. The whole number overflows if one of the following cases occurs:

- $c_{d-1} = 1$, i.e. the most significant digit overflows;
- $b_{d-1} = c_{d-2} = 1$, i.e. the second most significant digit overflows, and this propagates through the most significant digit;
- $b_{d-1} = b_{d-2} = c_{d-3} = 1$, i.e. the third most significant digit overflows, and this propagates through the two most significant digits;
- etc.
- $b_{d-1} = b_{d-2} = \cdots = b_1 = c_0 = 1$, i.e. the least significant digit overflows, and this propagates through all digits.

Note that all these cases are mutually exclusive, because each pair of them contains some position $i$, where one case has $c_i$ and the other one has $b_i$. Both $b_i$ and $c_i$ cannot be 1 at the same time.

Alg. 13 computes the weighted sum $m$ of all $b_i$ and $c_i$, such that this sum will be at least $2^{d-1}$ iff one of the above cases is true. We see that for each case given above, the weights given to the bits involved in this case sum up to exactly $2^{d-1}$. If some other bits are also set, then the sum can be greater than that.

It is also not difficult to see that if none of the cases hold, then the sum will be less than $2^{d-1}$. Indeed, suppose that $s$ is the greatest index where $c_s = 1$, and there exists some $t \in \{s+1, \ldots, d-1\}$, such that $b_t = 0$. Then the weighted sum can be at most

$$\sum_{i=0}^{s} 2^i + \sum_{i=s+1}^{d-1} 2^{i-1} - 2^{t-1} = 2^{s+1} - 1 + 2^{d-1} - 2^s - 2^{t-1} = 2^{d-1} - 1 + 2^s - 2^{t-1},$$

which is less than $2^{d-1}$, because $t - 1 \geq s$.

**Par:** Length $d$; "middle" ($\times 2$) and output moduli $Q, N, M$ satisfying $N > 2^d$; radix $r \leq (Q+1)/2$
**Input:** $\langle\!|v|\!\rangle^{(q)}$, $C \in \mathbb{N}$
**Output:** Bit $\langle\!|z|\!\rangle^{(M)}$ indicating whether $v < C$

$\langle\!|\mathbf{p}|\!\rangle^{(Q)} \leftarrow \mathsf{asymm\_rp}[Q, r, d](\langle\!|v|\!\rangle)$
$\mathsf{S}$: $w_\mathsf{S} \leftarrow \mathsf{split}^{-1}([\![\mathbf{p}]\!]_\mathsf{S})$
$\langle\!|\mathbf{a}|\!\rangle^{(Q)} \leftarrow \langle\!|\mathbf{p}|\!\rangle$; $[\![\mathbf{a}]\!]_\mathsf{S} := \mathsf{split}(w_\mathsf{S} + r^d - q)$
$\langle\!|\mathbf{b}|\!\rangle^{(Q)} \leftarrow \langle\!|\mathbf{p}|\!\rangle$; $[\![\mathbf{b}]\!]_\mathsf{S} := \mathsf{split}(((w_\mathsf{S} - C) \bmod q) + r^d - q)$
$\langle\!|c|\!\rangle^{(M)} \leftarrow \langle\!|0|\!\rangle$; $[\![c]\!]_\mathsf{S} := w_\mathsf{S} \geq C \;?\; 1 : 0$
$\langle\!|x|\!\rangle^{(M)} \leftarrow \mathsf{long\_of}[d, Q, N, M, r](\langle\!|\mathbf{a}|\!\rangle)$
$\langle\!|y|\!\rangle^{(M)} \leftarrow \mathsf{long\_of}[d, Q, N, M, r](\langle\!|\mathbf{b}|\!\rangle)$
**return** $1 + \langle\!|x|\!\rangle - \langle\!|y|\!\rangle - \langle\!|c|\!\rangle$

**Algorithm 14:** Inequality of a private value and a constant: ineq

### 3.10 Inequality check

The inequality (wrt. a constant) is given in Alg. 14. Similarly to [2], we compute the overflows of the values $\langle\!|v|\!\rangle$ and $\langle\!|v|\!\rangle - C$, where the latter may roll over modulo $q$. We are interested in overflows wrt. $q$, but Alg. 13 computes them wrt. $r^d$, hence we adjust Server's share of $v$ by adding $r^d - q$ to it. This addition happens after we have split $\langle\!|v|\!\rangle$ to digits. We obtain the shares of digits of $\langle\!|v|\!\rangle - C$ similarly, doing the adjustments at server's side after having split $\langle\!|v|\!\rangle$ into digits and obtained Server's MACs on them. We use $N = 67$ and $M = 71$ as the moduli.

The correctness of Alg. 14 follows through the same arguments as the correctness of Attrapadung et al.'s inequality protocol [2]. Let us give a rehash of these arguments.

We have the number $v$ shared as $([\![v]\!]_\mathsf{P} + [\![v]\!]_\mathsf{S}) \bmod q$, and the constant $C$. Alg. 14 applies the asymmetric reshare to parts to it; if we ignore the plurality of digits, then we can think of this operation as *resharing* $[\![v]\!]$: we change $[\![v]\!]_\mathsf{P}$ and $[\![v]\!]_\mathsf{S}$ (now denoted $w_\mathsf{S}$ in Alg. 14), such that they are still the shares of $v$. Let us write

- $v_1 := [\![v]\!]_\mathsf{P}$;
- $v_2 := [\![v]\!]_\mathsf{S}$;
- $v_2' := ([\![v]\!]_\mathsf{S} - C) \bmod q$, meaning that either $v_2' = v_2 - C$ or $v_2' = v_2 + q - C$.

If we ignore the shift $r^d - q$, and the plurality of digits, then we can think that the shares of $[\![\mathbf{a}]\!]$ are $v_1$ and $v_2$, while the shares of $[\![\mathbf{b}]\!]$ are $v_1$ and $v_2'$. When computing the overflows, we figure out the bit $x$ indicating whether $v_1 + v_2 \geq q$. Similarly, the bit $y$ indicates whether $v_1 + v_2' \geq q$. We also have the bit $c$ indicating whether $v_2 \geq C$. It turns out that knowing the values of these three bits is sufficient to figure out whether $v < C$. Let us consider all eight possibilities for the values of the bits $x, y, c$, and figure out the output bit of the inequality check in each case.

**000** $v_1 + v_2 < q$, $v_1 + v_2' < q$, $v_2 < C$. In this case, $v_2' = v_2 + q - C$. We have $C = v_2 - v_2' + q > v_2 - v_2' + v_1 + v_2' = v_1 + v_2$, i.e. the result should be 1.

**001** $v_1 + v_2 < q$, $v_1 + v_2' < q$, $v_2 \geq C$. In this case, $v_2' = v_2 - C$. We have $0 \leq v_1 + v_2 - C < q$. We have $C \leq (v_1 + v_2) \bmod q$, i.e. the result should be 0.

**010** $v_1 + v_2 < q$, $v_1 + v_2' \geq q$, $v_2 < C$. In this case, $v_2' = v_2 + q - C$. We have $C = v_2 - v_2' + q \leq v_2 - v_2' + v_1 + v_2' = v_1 + v_2$, i.e. the result should be 0.

**011** $v_1 + v_2 < q$, $v_1 + v_2' \geq q$, $v_2 \geq C$. In this case, $v_2' = v_2 - C$. This case is impossible, because subtraction means that $v_2' \leq v_2$, but the first two inequalities mean that $v_2' > v_2$.

**100** $v_1 + v_2 \geq q$, $v_1 + v_2' < q$, $v_2 < C$. In this case, $v_2' = v_2 + q - C > v_2$, but also $v_2' < q - v_1 \leq v_2$. This case is impossible.

**101** $v_1 + v_2 \geq q$, $v_1 + v_2' < q$, $v_2 \geq C$. In this case, $v_2' = v_2 - C$. We have $C = v_2 - v_2' > v_2 - q + v_1$, i.e. the result should be 1.

**110** $v_1 + v_2 \geq q$, $v_1 + v_2' \geq q$, $v_2 < C$. In this case, $v_2' = v_2 + q - C$. We have $C = v_2 - v_2' + q > v_2 - q + v_1$ (because $v_2' < q$ and $q > v_1$), i.e. the result should be 1.

**111** $v_1 + v_2 \geq q$, $v_1 + v_2' \geq q$, $v_2 \geq C$. In this case, $v_2' = v_2 - C$. We have $C = v_2 - v_2' \leq v_2 - q + v_1$, i.e. the result should be 0.

We see that in all six cases that are actually possible, the result is $1 + x - y - c$.

Alg. 14 allows us to perform rejection checks on $\langle\langle \mathbf{z} \rangle\rangle$ and $\langle\langle \mathbf{x} \rangle\rangle$, as shown in Alg. 15, albeit it provides only privacy, but no correctness against a malicious Server. Moreover, when the result of rejection check is opened, it may give Server some information on $\mathbf{z}$ and $\mathbf{x}$. Rejection check with AP-security is still useful: if the declassified result of rejection check is "OK", then we first declassify $\mathbf{z}$ only to the Phone. The Phone verifies the signature $(\mathbf{z}, c)$ and only on success sends it to the Server. If the Server interfered with the computations and caused the rejection check result to be "OK", but $(\mathbf{z}, c)$ does not verify, then the Phone recognizes that the Server is malicious. If the Server caused the rejection check result to be "not OK", even though it should have been "OK", then the Server only reduced the amount of information it would have learned: if the Server hadn't tampered with the computations, it would have learned the entire $\mathbf{z}$ (and also $\mathbf{x}$; it can be computed from the signature and the public key) from the signature. If the Server tampered with the computations of the rejection check in a manner that didn't change its result, then it would have received at least as much information about $\mathbf{z}$ and $\mathbf{x}$ also by not tampering. Hence the Server's best tactic is to not tamper.

Alg. 15 invokes the ineq protocol (with parameters $d, Q, N, M, r$), and then computes the long conjunction over the results. We see that it adds up the negations of the bits, and then returns whether the result is 0. Actually, the vector $\mathbf{v}$ is much longer than the modulus $M$, hence the sum may wrap around. The Phone will consider this wraparound as Server's malicious behaviour. We have chosen the modulus $M = 71$ so, that probability of the sum being at least $M$ is less than $2^{-256}$ if both parties behave honestly. Alg. 15 does not declassify its output; the declassification is part of $\Pi_{\mathsf{ML\text{-}DSA}}$.

**Par:** ML-DSA parameters, and $d, Q, N, M, r$ (same as in Alg. 14)
**Input:** $\left(\sum_{t=0}^{255} \langle\!\langle z_{i-1,t} \rangle\!\rangle \cdot X^t\right)_{i=1}^{\ell}$, $\left(\sum_{t=0}^{255} \langle\!\langle x_{i-1,t} \rangle\!\rangle \cdot X^t\right)_{i=1}^{k}$
**Output:** A bit indicating whether rejection check is passed

**foreach** $i \in [\ell]$, $t \in [256]$ **do**
  $\langle\!\langle v_{256 \cdot i + t} \rangle\!\rangle^{(M)} \leftarrow \mathsf{ineq}[d, Q, N, M, r](\langle\!\langle z_{i,t} \rangle\!\rangle + \gamma_1 - \beta - 1, 2(\gamma_1 - \beta) - 1)$
**foreach** $i \in [k]$, $t \in [256]$ **do**
  $\langle\!\langle v_{256 \cdot (i+\ell) + t} \rangle\!\rangle^{(M)} \leftarrow \mathsf{ineq}[d, Q, N, M, r](\langle\!\langle x_{i,t} \rangle\!\rangle + \gamma_2 - \beta - 1, 2(\gamma_2 - \beta) - 1)$
$\langle\!\langle s \rangle\!\rangle^{(M)} \leftarrow \sum_{i=0}^{\mathsf{length}(\mathbf{v})-1} (1 - \langle\!\langle v_i \rangle\!\rangle^{(M)})$
$\langle\!\langle \mathbf{b} \rangle\!\rangle \leftarrow \mathsf{ch\_vec}[M, 2](\langle\!\langle s \rangle\!\rangle^{(M)})$
**return** $\langle\!\langle b_0 \rangle\!\rangle$

**Algorithm 15:** Rejection check: rej_check

**Par:** ML-DSA parameters, and $d, Q, N, M, r$
**Input:** $\mathbf{A}$, $\mathbf{t}$, $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle$, $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle$, $\mu$
**Output:** Signature $(\mathbf{z}, c)$ on $\mu$, or $\perp$

$\langle\!\langle \mathbf{y} \rangle\!\rangle, \langle\!\langle \mathbf{w}^H \rangle\!\rangle \leftarrow \mathsf{gcom}[k, \ell, \alpha, \gamma_1](\mathbf{A})$
$\langle\!\langle \mathbf{w}^L \rangle\!\rangle \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle - \alpha \cdot \langle\!\langle \mathbf{w}^H \rangle\!\rangle$
$c \leftarrow H(\mu, \mathsf{declassify}(\langle\!\langle \mathbf{w}^H \rangle\!\rangle))$
$\langle\!\langle \mathbf{z} \rangle\!\rangle = \left(\sum_{t=0}^{255} \langle\!\langle z_{i-1,t} \rangle\!\rangle \cdot X^t\right)_{i=1}^{\ell} \leftarrow \langle\!\langle \mathbf{y} \rangle\!\rangle + c \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle$
$\langle\!\langle \mathbf{x} \rangle\!\rangle = \left(\sum_{t=0}^{255} \langle\!\langle x_{i-1,t} \rangle\!\rangle \cdot X^t\right)_{i=1}^{k} \leftarrow \langle\!\langle \mathbf{w}^L \rangle\!\rangle - c \cdot \langle\!\langle \mathbf{s}_2 \rangle\!\rangle$
**if** $\mathsf{declassify}(\mathsf{rej\_check}(\langle\!\langle \mathbf{z} \rangle\!\rangle, \langle\!\langle \mathbf{x} \rangle\!\rangle))$ **then**
  $\mathsf{S} \rightarrow \mathsf{P}$: $[\![\mathbf{z}]\!]_{\mathsf{S}}$
  $\mathsf{P}$: $\mathbf{z} \leftarrow [\![\mathbf{z}]\!]_{\mathsf{S}} + [\![\mathbf{z}]\!]_{\mathsf{P}}$
  $\mathsf{P}$: **stop** if $\neg\mathsf{Verify}((\mathbf{A}, \mathbf{t}), \mu, (\mathbf{z}, c))$
  $\mathsf{P} \rightarrow \mathsf{S}$: $\mathbf{z}$
  **return** $(\mathbf{z}, c)$
**else return** $\perp$ ;

**Algorithm 16:** A signing attempt: sign_trilithium

### 3.11 Distributed signing

The entire signing attempt is given in Alg. 16.

## 4 Security of components

Each algorithm in Sec. 3 corresponds to a real system $\Pi$; the system $\Pi_{\mathsf{ML\text{-}DSA}}$ is the composition of the systems corresponding to Alg. 5 and Alg. 16. The security analysis becomes more modular when we state the ideal functionalities securely implemented by these systems.

Thus, for each of these algorithms, we will now describe an ideal functionality, a real protocol for running this functionality, and possibly another ideal functionality for generating the correlated randomness. We will then show that the real protocol is a secure implementation of the ideal functionality in the hybrid model that includes the correlated randomness generation, and all functionalities for the sub-protocols invoked by the algorithm. We start with the declassification

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{P}$ and $\boldsymbol{\Delta}_\mathsf{S}$. If party $\mathsf{X}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{X})$ to the adversary.
- On input $(\mathsf{declassify}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{declassify}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{S})$ from $\mathsf{S}$ (where $\langle\!\langle v \rangle\!\rangle_\mathsf{X} = ([\![v]\!]_\mathsf{X}, [\![M_\mathsf{S}]\!]_\mathsf{X}, [\![M_\mathsf{P}]\!]_\mathsf{X})$), proceed as follows:
  1. Notify $\mathcal{S}$. If nobody corrupt, send $(\mathsf{proceed?}, sid)$ to $\mathcal{S}$, get back $(\mathsf{proceed!}, sid)$. If $\mathsf{S}$ corrupt, send $(\mathsf{input}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{S})$ to $\mathcal{S}$ and get back $(\mathsf{update}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{S})$. If $\mathsf{P}$ corrupt, send $(\mathsf{input}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P}, [\![v]\!]_\mathsf{S}, [\![M_\mathsf{P}]\!]_\mathsf{S})$ to $\mathcal{S}$ and get back $(\mathsf{update}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P})$. Update the component of $\langle\!\langle v \rangle\!\rangle$ with the value received from $\mathcal{S}$.
  2. Compute $v \leftarrow [\![v]\!]_\mathsf{P} + [\![v]\!]_\mathsf{S}$. Let the boolean $b_\mathsf{P}$ [resp. $b_\mathsf{S}$] indicate whether $M_\mathsf{P}$ [resp. $M_\mathsf{S}$] verifies.
  3. Output the result. Proceed as follows:
     - If nobody corrupt: If $b_\mathsf{P}$, send $(\mathsf{result}, sid, v)$ to $\mathsf{P}$, otherwise send $(\mathsf{result}, sid, \bot)$ to $\mathsf{P}$. If $b_\mathsf{S} \wedge b_\mathsf{P}$, send $(\mathsf{result}, sid, v)$ to $\mathsf{S}$, otherwise send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.
     - If $\mathsf{P}$ corrupt: send $(\mathsf{result?}, sid)$ to $\mathcal{S}$. Receive back $(\mathsf{result!}, sid, v', b)$. Send $(\mathsf{result}, sid, v')$ to $\mathsf{P}$. If $b \wedge b_\mathsf{S}$ then send $(\mathsf{result}, sid, v)$ to $\mathsf{S}$, otherwise send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.
     - If $\mathsf{S}$ corrupt: If $\neg b_\mathsf{P}$, then send $(\mathsf{result}, sid, \bot)$ to $\mathsf{P}$, $\mathsf{S}$ and $\mathcal{S}$. Otherwise send $(\mathsf{result}, sid, v)$ to $\mathsf{P}$ and $(\mathsf{result?}, sid, v, b_\mathsf{S})$ to $\mathcal{S}$. Get back $(\mathsf{result}, sid, v')$ and forward it to $\mathsf{S}$.

**Functionality 2:** $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$: ideal functionality for declassification

from Sec. 3.1, then give the generic protocols and precomputation functionalities, as much as these can be given, then go through the algorithms in Sec. 3, and finish with the description of $\Pi_{\mathsf{ML\text{-}DSA}}$ and its security proof in Sec. 5.

### 4.1 Declassification

**Protocol and ideal functionality for AA security.** The ideal functionality for AA-secure declassification is given in Func. 2. Similarly to Alg. 3, it is asymmetric with respect of the roles of $\mathsf{P}$ and $\mathsf{S}$. We see that the steps of $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ are typical for ideal functionalities: it first allows the adversary to adjust the input(s) of corrupted party(-ies), then performs the actual computation (which, in this case consists of recovering $v$ and checking the MACs), then informs the adversary of the result and allows it to control when the result is sent back and which result is sent back to corrupted parties. In Func. 2 and below, we use $\mathsf{X}$ to denote either $\mathsf{S}$ or $\mathsf{P}$; the functionality allows one of the parties to be corrupted.

The description of Alg. 3 can be turned to the description of the real system consisting of machines $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$; this description is given in Prot. 1. The machines $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$ follow this description when not corrupted, and act as described in Sec. 2.5 when corrupted. To show that Prot. 1 securely implements

- On input $(\mathsf{corrupted}, sid)$ from $\mathcal{A}$ to $\mathcal{M}_\mathsf{X}$ before the initialization, machine $\mathcal{M}_\mathsf{X}$ sends $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$ and becomes corrupted.
- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{X})$ from $\mathsf{X}$ to $\mathcal{M}_\mathsf{X}$: store $\boldsymbol{\Delta}_\mathsf{X}$. If $\mathcal{M}_\mathsf{X}$ is corrupt, send $(\mathsf{globkeys}, sid, \boldsymbol{\Delta}_\mathsf{X})$ to $\mathcal{A}$.
- On input $(\mathsf{declassify}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{S})$ from $\mathsf{S}$ to $\mathcal{M}_\mathsf{S}$, the machine $\mathcal{M}_\mathsf{S}$ works as follows:
  - Send $(\mathsf{msg1}, sid, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}, {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{S})$ to $\mathcal{M}_\mathsf{P}$.
  - Expect back $(\mathsf{msg2}, sid, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P}, {}_\mathsf{ms}\langle\!\langle v \rangle\!\rangle_\mathsf{P})$.
    - If the message was $(\mathsf{msg2}, sid, \bot)$, then send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$ and do not proceed further.
  - Recover $v \leftarrow {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P} + {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}$.
  - Verify $\mathsf{S}$'s MAC: check that ${}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P} \cdot \boldsymbol{\Delta}_\mathsf{S} = {}_\mathsf{ms}\langle\!\langle v \rangle\!\rangle_\mathsf{P} + {}_\mathsf{ms}\langle\!\langle v \rangle\!\rangle_\mathsf{S}$
  - If the MAC verifies, send $(\mathsf{result}, sid, v)$ to $\mathsf{S}$.
  - If the MAC does not verify, send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.
- On input $(\mathsf{declassify}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P})$ from $\mathsf{P}$ to $\mathcal{M}_\mathsf{P}$, the machine $\mathcal{M}_\mathsf{P}$ works as follows:
  - Expect $(\mathsf{msg1}, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}, {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{S})$ from $\mathcal{M}_\mathsf{S}$.
  - Recover $v \leftarrow {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P} + {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}$.
  - Verify $\mathsf{P}$'s MAC: check that ${}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S} \cdot \boldsymbol{\Delta}_\mathsf{P} = {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{P} + {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{S}$
  - If the MAC verifies, send $(\mathsf{msg2}, sid, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P}, {}_\mathsf{ms}\langle\!\langle v \rangle\!\rangle_\mathsf{P})$ to $\mathcal{M}_\mathsf{S}$ and $(\mathsf{result}, sid, v)$ to $\mathsf{P}$.
  - If the MAC does not verify, send $(\mathsf{msg2}, sid, \bot)$ to $\mathcal{M}_\mathsf{S}$ and $(\mathsf{result}, sid, \bot)$ to $\mathsf{P}$.

**Protocol 1:** AA declassification protocol

Func. 2, we construct the simulator. As the corruptions of parties are static (may only happen before the actual execution starts), our simulator is basically a choice between three simulators: one for the case with not corrupted parties, one for corrupted Phone, and one for corrupted Server. The first case is trivial; the simulator internally executes both $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$ and relays the scheduling information between $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ and $\mathcal{A}$. Let us describe the construction of the other two simulators.

**Simulator for AA-secure declassification with corrupted Phone.** During initialization, the simulator receives $\boldsymbol{\Delta}_\mathsf{P}$. During declassification, it receives $(\mathsf{input}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P}, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}, {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{S})$ from the ideal functionality and tells the adversary that the share $\langle\!\langle v \rangle\!\rangle_\mathsf{P}$ is being declassified (as if $\mathcal{M}_\mathsf{P}$ reporting that it got the $\mathsf{declassify}$-command from the environment). It creates and sends to $\mathcal{A}$ the message $(\mathsf{msg1}, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{S}, {}_\mathsf{mp}\langle\!\langle v \rangle\!\rangle_\mathsf{S})$, as if $\mathcal{M}_\mathsf{P}$ was reporting to $\mathcal{A}$ that it received this message from $\mathcal{M}_\mathsf{S}$.

If the adversary sends back (as the command to $\mathcal{M}_\mathsf{P}$ to send a message to $\mathcal{M}_\mathsf{S}$) the message $(\mathsf{msg2}, sid, \bot)$, followed by the command to $\mathcal{M}_\mathsf{P}$ to output $v'$ to $\mathsf{P}$, then the simulator sends the message $(\mathsf{update}, sid, \langle\!\langle v \rangle\!\rangle_\mathsf{P})$ back to the ideal functionality, waits to receive $(\mathsf{result?}, sid)$ from the ideal functionality, and sends back $(\mathsf{result!}, sid, v', \mathsf{false})$ to the ideal functionality.

But if the message $(\mathsf{msg2}, sid, {}_\mathsf{v}\langle\!\langle v \rangle\!\rangle_\mathsf{P}, {}_\mathsf{ms}\langle\!\langle v \rangle\!\rangle_\mathsf{P})$, intended for $\mathcal{M}_\mathsf{P}$ to send it to $\mathcal{M}_\mathsf{S}$, is sent by $\mathcal{A}$ to the simulator, again followed by $v'$ for $\mathsf{P}$, then update $\langle\!\langle v \rangle\!\rangle_\mathsf{P}$

with the components received from $\mathcal{A}$ and send $(\mathsf{update}, sid, \langle\!\langle v \rangle\!\rangle_{\mathsf{P}})$ back to the ideal functionality. Wait to receive $(\mathsf{result?}, sid)$ from the ideal functionality, and send back $(\mathsf{result!}, sid, v', \mathsf{true})$ to the ideal functionality.

**Simulator for AA-secure declassification with corrupted Server.** During initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$. During declassification, it receives $(\mathsf{input},$
$sid, \langle\!\langle v \rangle\!\rangle_{\mathsf{S}})$ from the ideal functionality and tells the adversary that the share $\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}$ is being declassified. The simulator then expects to receive from the adversary (as $\mathcal{M}_{\mathsf{S}}$ sending it to $\mathcal{M}_{\mathsf{P}}$) the message $(\mathsf{msg1}, sid, {}_{\mathsf{v}}\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}, {}_{\mathsf{mp}}\langle\!\langle v \rangle\!\rangle_{\mathsf{S}})$. The simulator updates $\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}$ with the components it received and sends $(\mathsf{update}, sid, \langle\!\langle v \rangle\!\rangle_{\mathsf{S}})$ back to the ideal functionality.

Wait for the next message from the ideal functionality. If it is $(\mathsf{result}, sid, \bot)$, then send $(\mathsf{msg2}, sid, \bot)$ to the adversary (as message from $\mathcal{M}_{\mathsf{P}}$ to $\mathcal{M}_{\mathsf{S}}$). If the message is $(\mathsf{result?}, sid, v, b_{\mathsf{S}})$, then compute

- ${}_{\mathsf{v}}\langle\!\langle v \rangle\!\rangle_{\mathsf{P}} \leftarrow v - {}_{\mathsf{v}}\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}$
- ${}_{\mathsf{ms}}\langle\!\langle v \rangle\!\rangle_{\mathsf{P}} \leftarrow \boldsymbol{\Delta}_{\mathsf{S}} \cdot {}_{\mathsf{v}}\langle\!\langle v \rangle\!\rangle_{\mathsf{P}} - {}_{\mathsf{ms}}\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}$

and send $(\mathsf{msg2}, sid, {}_{\mathsf{v}}\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}, {}_{\mathsf{ms}}\langle\!\langle v \rangle\!\rangle_{\mathsf{P}})$ to $\mathcal{A}$ (as $\mathcal{M}_{\mathsf{S}}$ reporting receiving a message from $\mathcal{M}_{\mathsf{P}}$). If $b_{\mathsf{S}} = \mathsf{true}$ then send $(\mathsf{result}, sid, v)$ to the ideal functionality, otherwise send $(\mathsf{result}, sid, \bot)$ to the ideal functionality.

**Protocol and ideal functionality for AP security.** In Sec. 3.1 we have already discussed how the protocol for AP-secure declassification would look like: it would remove some message parts and checks from Alg. 3. Let us now give the description explicitly in Func. 3 and Prot. 2. We notice that the ideal functionality still ensures that if $\mathsf{S}$ is honest, then it learns either the to-be-declassified value $v$ or nothing at all. But if $\mathsf{S}$ is corrupt, then the adversary $\mathcal{S}$ can freely add a change $\hat{v}$ to the value that is received by $\mathsf{P}$. Indeed, after $\mathcal{S}$ has received the $(\mathsf{input}, sid, \langle\!|v|\!\rangle_{\mathsf{S}})$-message, it has to add $\hat{v}$ to the $\mathsf{v}$-component of $\langle\!|v|\!\rangle_{\mathsf{S}}$ in the $\mathsf{update}$-message to impose the same change to $v$. This change does not affect the validity of any MACs.

**Simulator for AP-secure declassification with corrupted Phone.** During initialization, the simulator does not receive anything. During declassification, it receives $(\mathsf{input}, sid, \langle\!|v|\!\rangle_{\mathsf{P}}, {}_{\mathsf{v}}\langle\!|v|\!\rangle_{\mathsf{S}})$ from the ideal functionality and tells the adversary that the share $\langle\!|v|\!\rangle_{\mathsf{P}}$ is being declassified (as if $\mathcal{M}_{\mathsf{P}}$ reporting that it got the $\mathsf{declassify}$-command from $\mathsf{P}$). It creates the message $(\mathsf{msg1}, {}_{\mathsf{v}}\langle\!|v|\!\rangle_{\mathsf{S}})$ and sends it to $\mathcal{A}$ (as $\mathcal{M}_{\mathsf{P}}$ reporting to $\mathcal{A}$ that it received this message from $\mathcal{M}_{\mathsf{S}}$).

If the adversary sends back (as the command to $\mathcal{M}_{\mathsf{P}}$ to send a message to $\mathcal{M}_{\mathsf{S}}$) the message $(\mathsf{msg2}, sid, \bot)$, followed by the command to $\mathcal{M}_{\mathsf{P}}$ to output $v'$ to $\mathsf{P}$, then the simulator sends the message $(\mathsf{update}, sid, \langle\!|v|\!\rangle_{\mathsf{P}})$ back to the ideal functionality, waits to receive $(\mathsf{result?}, sid)$ from the ideal functionality, and sends back $(\mathsf{result!}, sid, v', \mathsf{false})$ to the ideal functionality.

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid)$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{S}$. If $\mathsf{S}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{S})$ to the adversary.
- On input $(\mathsf{declassify}, sid, \langle\!|v|\!\rangle_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{declassify}, sid, \langle\!|v|\!\rangle_\mathsf{S})$ from $\mathsf{S}$, proceed as follows:
  1. Notify $\mathcal{S}$.
     - If nobody is corrupt, send $(\mathsf{proceed?}, sid)$ to $\mathcal{S}$, get back $(\mathsf{proceed!}, sid)$.
     - If $\mathsf{P}$ is corrupt, send $(\mathsf{input}, sid, \langle\!|v|\!\rangle_\mathsf{P}, [\![v]\!]_\mathsf{S})$ to $\mathcal{S}$ and get back $(\mathsf{update}, sid, \langle\!|v|\!\rangle_\mathsf{P})$.
     - If $\mathsf{S}$ is corrupt, send $(\mathsf{input}, sid, \langle\!|v|\!\rangle_\mathsf{S})$ to $\mathcal{S}$ and get back $(\mathsf{update}, sid, \langle\!|v|\!\rangle_\mathsf{S})$.

     Update the component of $\langle\!|v|\!\rangle$ with the value received from $\mathcal{S}$.
  2. Recover $v$ from $\langle\!|v|\!\rangle$. Let the boolean $b_\mathsf{S}$ indicate whether ${}_{\mathsf{ms}}\langle\!|v|\!\rangle$ verifies.
  3. Output the result. Proceed as follows:
     - If nobody is corrupt: Send $(\mathsf{result}, sid, v)$ to $\mathsf{P}$. If $b_\mathsf{S}$, send it to $\mathsf{S}$, too, otherwise send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.
     - If $\mathsf{S}$ is corrupt: send $(\mathsf{result?}, sid, v, b_\mathsf{S})$ to $\mathcal{S}$ and $(\mathsf{result}, sid, v)$ to $\mathsf{P}$. Let $\mathcal{S}$ send back $(\mathsf{result}, sid, v')$. Send $(\mathsf{result}, sid, v')$ to $\mathsf{S}$.
     - If $\mathsf{P}$ is corrupt: send $(\mathsf{result?}, sid)$ to $\mathcal{S}$. Receive back $(\mathsf{result!}, sid, v', b)$. Send $(\mathsf{result}, sid, v')$ to $\mathsf{P}$. If $b \wedge b_\mathsf{S}$, then send $(\mathsf{result}, sid, v)$ to $\mathsf{S}$, otherwise send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.

**Functionality 3:** AP-secure declassification

- On input $(\mathsf{corrupted}, sid)$ from $\mathcal{A}$ to $\mathcal{M}_\mathsf{X}$ before the initialization, machine $\mathcal{M}_\mathsf{X}$ sends $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$ and becomes corrupted.
- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$ to $\mathcal{M}_\mathsf{S}$: store $\boldsymbol{\Delta}_\mathsf{S}$. If $\mathcal{M}_\mathsf{S}$ is corrupt, send $(\mathsf{globkeys}, sid, \boldsymbol{\Delta}_\mathsf{S})$ to $\mathcal{A}$.
- On input $(\mathsf{init}, sid)$ from $\mathsf{P}$ to $\mathcal{M}_\mathsf{P}$: (nothing).
- On input $(\mathsf{declassify}, sid, \langle\!|v|\!\rangle_\mathsf{S})$ from $\mathsf{S}$ to $\mathcal{M}_\mathsf{S}$, the machine $\mathcal{M}_\mathsf{S}$ works as follows:
  - Send $(\mathsf{msg1}, sid, {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{S})$ to $\mathcal{M}_\mathsf{P}$.
  - Expect back $(\mathsf{msg2}, sid, \langle\!|v|\!\rangle_\mathsf{P})$.
    - If the message was $(\mathsf{msg2}, sid, \bot)$, then send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$ and do not proceed further.
  - Recover $v \leftarrow {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{P} + {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{S}$.
  - Verify $\mathsf{S}$'s MAC: check that ${}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{P} \cdot \boldsymbol{\Delta}_\mathsf{S} = {}_{\mathsf{ms}}\langle\!|v|\!\rangle_\mathsf{P} + {}_{\mathsf{ms}}\langle\!|v|\!\rangle_\mathsf{S}$
  - If the MAC verifies, send $(\mathsf{result}, sid, v)$ to $\mathsf{S}$.
  - If the MAC does not verify, send $(\mathsf{result}, sid, \bot)$ to $\mathsf{S}$.
- On input $(\mathsf{declassify}, sid, \langle\!|v|\!\rangle_\mathsf{P})$ from $\mathsf{P}$ to $\mathcal{M}_\mathsf{P}$, the machine $\mathcal{M}_\mathsf{P}$ works as follows:
  - Expect $(\mathsf{msg1}, {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{S})$ from $\mathcal{M}_\mathsf{S}$.
  - Recover $v \leftarrow {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{P} + {}_{\mathsf{v}}\langle\!|v|\!\rangle_\mathsf{S}$.
  - Send $(\mathsf{msg2}, sid, \langle\!|v|\!\rangle_\mathsf{P})$ to $\mathcal{M}_\mathsf{S}$ and $(\mathsf{result}, sid, v)$ to $\mathsf{P}$.

**Protocol 2:** AP declassification protocol

But if the adversary sends back the message $(\mathsf{msg2}, sid, \langle\!|v|\!\rangle_{\mathsf{P}})$ for $\mathcal{M}_{\mathsf{S}}$, again followed by $v'$ for $\mathsf{P}$, then the simulator updates $\langle\!|v|\!\rangle_{\mathsf{P}}$ and sends $(\mathsf{update}, sid, \langle\!|v|\!\rangle_{\mathsf{P}})$ back to the ideal functionality. Waits to receive $(\mathsf{result?}, sid)$ from the ideal functionality, and sends back $(\mathsf{result!}, sid, v', \mathsf{true})$ to the ideal functionality.

**Simulator for AP-secure declassification with corrupted Server.** During initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$. During declassification, it receives $(\mathsf{input}, sid, \langle\!|v|\!\rangle_{\mathsf{S}})$ from the ideal functionality and tells the adversary that the share $\langle\!|v|\!\rangle_{\mathsf{S}}$ is being declassified.

The simulator expects $\mathcal{A}$ to tell $\mathcal{M}_{\mathsf{S}}$ to send the first message $(\mathsf{msg1}, sid, [\![v']\!]_{\mathsf{S}})$ to $\mathcal{M}_{\mathsf{P}}$. The simulator then sends to the ideal functionality the message $(\mathsf{update}, sid, ([\![v']\!]_{\mathsf{S}}, {}_{\mathsf{ms}}\langle\!|v|\!\rangle_{\mathsf{S}}))$, effectively changing the to-be-declassified value by $\varepsilon = [\![v']\!]_{\mathsf{S}} - [\![v]\!]_{\mathsf{S}}$.

The ideal functionality sends $(\mathsf{result?}, sid, v, b_{\mathsf{S}})$ to the simulator. The simulator computes

- $[\![v]\!]_{\mathsf{P}} \leftarrow v - [\![v]\!]_{\mathsf{S}}$
- ${}_{\mathsf{ms}}\langle\!|v|\!\rangle_{\mathsf{P}} \leftarrow \boldsymbol{\Delta}_{\mathsf{S}} \cdot [\![v]\!]_{\mathsf{P}} - {}_{\mathsf{ms}}\langle\!|v|\!\rangle_{\mathsf{S}}$

and send $(\mathsf{msg2}, sid, \langle\!|v|\!\rangle_{\mathsf{P}})$ to $\mathcal{A}$ (as if $\mathcal{M}_{\mathsf{S}}$ notifying $\mathcal{A}$ about the message it received from $\mathcal{M}_{\mathsf{P}}$). The simulator gets $v'$ from the adversary (as the value that $\mathsf{S}$ should receive) and sends $(\mathsf{result}, sid, v')$ to the ideal functionality.

### 4.2 Ideal functionalities corresponding to algorithms

The ideal functionalities for all AA-secure protocols above are instances of the generic functionality given in Func. 4. The functionality is parametrized by the function $f$ it is computing, which may be probabilistic. The functionality is also parametrized by the privacy levels of its inputs: parts of the inputs are secret-shared, while others are available in the open. The outputs are always private; this is the reason we did not declassify the outputs of Alg. 5 or Alg. 9. There is another constraint on outputs, less visible in Func. 4: the outputs may not be linearly dependent on other outputs or inputs. For this reason, $\mathbf{A} \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle$ was computed in both Alg. 9 and Alg. 16, and $\langle\!\langle \mathbf{t} \rangle\!\rangle$ was not computed in Alg. 5. Finally, a predicate $\mathbf{B}$ limiting the domain of $f$ is a parameter of the functionality.

Not all AA-secure protocols in Sec. 3 use all features of $f$ in Func. 4. Clearly, if a protocol is generating random values, then the corresponding $f$ is randomized. The function $f$ for "symmetric reshare to parts" is also randomized: it is resharing either $v$ or $v + q$ depending on whether $r \leq v$ or not, for a random $r \in \mathbb{Z}_q$. The predicate $\mathbf{B}$ is materially used only in the functionality describing the "zero check" (Alg. 6); for all others, $\mathbf{B} \equiv 1$.

The ideal functionalities for all AP-secure protocols, *except* for Alg. 10, are instances of the generic functionality given in Func. 5. Alg. 10 is not an instance, because the shares of the outputs of that protocol are not uniformly distributed over $\mathbb{Z}_Q$, and thus cannot be described in the manner allowed by Func. 5. Its corresponding ideal functionality will be given in Func. 8.

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{P}$ and $\boldsymbol{\Delta}_\mathsf{S}$. If party $\mathsf{X}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{X})$ to the adversary.
- On input $(\mathsf{compute}, sid, \langle\!\langle v^\mathsf{priv} \rangle\!\rangle_\mathsf{P}, v^\mathsf{pub})$ from $\mathsf{P}$ and $(\mathsf{compute}, sid, \langle\!\langle v^\mathsf{priv} \rangle\!\rangle_\mathsf{S}, v^\mathsf{pub})$ from $\mathsf{S}$, do the following:
  - If a party $\mathsf{X}$ is corrupted, send $v^\mathsf{pub}$ to $\mathcal{S}$. Also send $\langle\!\langle v^\mathsf{priv} \rangle\!\rangle_\mathsf{X}$ to the $\mathcal{S}$, and receive back an adjusted version of $\langle\!\langle v^\mathsf{priv} \rangle\!\rangle_\mathsf{X}$.
  - Recover $v^\mathsf{priv}$ from the shares; check the MACs. If an honest party's MAC does not verify, then stop.
  - Generate random $r$ of the type expected by $f$.
  - If $\mathbf{B}(v^\mathsf{priv}, v^\mathsf{pub}) = 0$, then send $(v^\mathsf{priv}, v^\mathsf{pub})$ to $\mathcal{S}$ and get back $w$. Otherwise compute $w = f(v^\mathsf{priv}, v^\mathsf{pub}, r)$.
  - Randomly secret-share $w$ (using $\boldsymbol{\Delta}_\mathsf{P}$ and $\boldsymbol{\Delta}_\mathsf{S}$), resulting in $\langle\!\langle w \rangle\!\rangle$.
  - If $\mathsf{X}$ is corrupted, then send $\langle\!\langle w \rangle\!\rangle_\mathsf{X}$ to $\mathcal{S}$.
  - Wait for the adversary to permit to proceed; also get updated value of $\langle\!\langle w \rangle\!\rangle_\mathsf{X}$.
  - Send $(\mathsf{result}, sid, \langle\!\langle w \rangle\!\rangle_\mathsf{S})$ to $\mathsf{S}$ and $(\mathsf{result}, sid, \langle\!\langle w \rangle\!\rangle_\mathsf{P})$ to $\mathsf{P}$.

**Functionality 4:** Generic AA-secure functionality for the function $f$

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid)$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{S}$. If $\mathsf{S}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{S})$ to the adversary.
- On input $(\mathsf{compute}, sid, \langle v^\mathsf{priv} \rangle_\mathsf{P}, v^\mathsf{pub})$ from $\mathsf{P}$ and $(\mathsf{compute}, sid, \langle v^\mathsf{priv} \rangle_\mathsf{S}, v^\mathsf{pub})$ from $\mathsf{S}$, do the following:
  - If a party $\mathsf{X}$ is corrupted, send $v^\mathsf{pub}$ to $\mathcal{S}$. Also send $\langle v^\mathsf{priv} \rangle_\mathsf{X}$ to the adversary, and receive back an adjusted version of $\langle v^\mathsf{priv} \rangle_\mathsf{X}$.
  - Recover $v^\mathsf{priv}$ from the shares; check the MACs. If $\mathsf{S}$ is honest, but the MACs does not verify, then stop.
  - Generate random $r$ of the type expected by $f$.
  - If $\mathsf{S}$ is honest, then compute $w = f(v^\mathsf{priv}, v^\mathsf{pub}, r)$.
  - If $\mathsf{S}$ is corrupted, then randomly generate the output share $\langle w \rangle_\mathsf{S}$ and send it to the adversary.
  - If $\mathsf{S}$ is corrupted, then receive the description of an algorithm $\mathfrak{A}$ from the adversary, compute $w = \mathfrak{A}(\langle v^\mathsf{priv} \rangle_\mathsf{P}, r)$; it must have the correct format.
  - Compute a sharing of $w$, giving $\langle w \rangle$. If $\langle w \rangle_\mathsf{S}$ already exists, then compute $\langle w \rangle_\mathsf{P}$ according to it and $w$ and $\boldsymbol{\Delta}_\mathsf{S}$.
  - If $\mathsf{X}$ is corrupted, then send $\langle w \rangle_\mathsf{X}$ to $\mathcal{S}$. Wait for the adversary's permission to proceed; also get updated $\langle w \rangle_\mathsf{X}$.
  - Send $(\mathsf{result}, sid, \langle w \rangle_\mathsf{S})$ to $\mathsf{S}$ and $(\mathsf{result}, sid, \langle w \rangle_\mathsf{P})$ to $\mathsf{P}$.

**Functionality 5:** Generic AP-secure functionality for the function $f$

- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{P}$ and $\boldsymbol{\Delta}_\mathsf{S}$. If party $\mathsf{X}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{X})$ to the adversary.
  - For AP-secure functionalities, the message from $\mathsf{P}$ does not contain $\boldsymbol{\Delta}_\mathsf{P}$.
- On input $(\mathsf{crreq}, sid)$ from both $\mathsf{P}$ and $\mathsf{S}$, generate the random values according to the desired distribution. Let $\mathbf{R}_\mathsf{P}$ be the values meant for $\mathsf{P}$, and $\mathbf{R}_\mathsf{S}$ the values meant for $\mathsf{S}$.
- If no party is corrupted, then send $(\mathsf{proceed?}, sid)$ to $\mathcal{S}$. If party $\mathsf{X}$ is corrupted, then send $(\mathsf{cr}, sid, \mathbf{R}_\mathsf{X})$ to $\mathcal{S}$.
- Wait for $\mathcal{S}$ to send back $(\mathsf{proceed!}, sid)$ (if no party is corrupted) or $(\mathsf{update}, sid, \mathbf{R}'_\mathsf{X})$ (if party $\mathsf{X}$ is corrupted). In the latter case, update $\mathbf{R}_\mathsf{X} := \mathbf{R}'_\mathsf{X}$.
- Send $(\mathsf{cr}, sid, \mathbf{R}_\mathsf{P})$ to $\mathsf{P}$ and $(\mathsf{cr}, sid, \mathbf{R}_\mathsf{S})$ to $\mathsf{S}$.

**Functionality 6:** $\mathcal{F}_{\mathsf{Alg}}^{\mathrm{offl}}$: Generic ideal functionality for precomputations in an algorithm "Alg"

We see that Func. 5 is parametrized similarly to Func. 4, except we have no use for the predicate $\mathbf{B}$. Note how we model the passive security (but active privacy) against $\mathsf{S}$: if $\mathsf{S}$ is corrupted then it chooses the output of the functionality on the basis of inputs, but cannot learn anything. The input to $\mathfrak{A}$ includes everything that $\mathcal{S}$ does not yet know: the values $\langle\!\langle v^{\mathsf{priv}} \rangle\!\rangle_\mathsf{S}$ and $v^{\mathsf{pub}}$ may be included in the description of $\mathfrak{A}$.

### 4.3 Defining precomputations

Certain algorithms in Sec. 3 (Alg. 4, Alg. 6, Alg. 7, Alg. 10, and Alg. 11) require the pre-distribution of values coming from a certain (joint) probability distribution. In the UC framework, we model the generation of such values as the availability of a certain ideal functionality, given in Func. 6. We see that it is initialized by the MAC keys. Indeed, the correlated random values may need MACs on them. The only protocol-specific part in this functionality is the actual sampling of random values.

### 4.4 Protocols from Algorithms

In Sec. 3, we have described our signing protocol and its subprotocols in the form of pseudocode typically used to present algorithms on top of MPC protocol sets. Each one of these algorithms could alternatively be represented as a real system in the UC framework, consisting of the machine $\mathcal{M}_\mathsf{P}$, machine $\mathcal{M}_\mathsf{S}$, and zero or more ideal components $\mathcal{F}$; the description of such a real system would make explicit certain details that are not mentioned in the algorithms.

A real system executing a protocol $\Pi_{\mathsf{Alg}}$ (given in Prot. 3) for an algorithm $\mathsf{Alg}$ contains $\mathcal{M}_\mathsf{P}$, $\mathcal{M}_\mathsf{S}$, the functionalities $\mathcal{F}_{\mathsf{sub}}$ for all subroutines that $\mathsf{Alg}$ invokes (except for subroutines that involve only local computations), and $\mathcal{F}_{\mathsf{Alg}}^{\mathrm{offl}}$ if $\mathsf{Alg}$ uses its own correlated randomness. Note that declassification is also realized by a functionality; protocols with AA security include $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AA}}$ while protocols with AP security include $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AP}}$.

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before any other commands, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{X})$ from $\mathsf{X}$ to machine $\mathcal{M}_\mathsf{X}$, store $\boldsymbol{\Delta}_\mathsf{X}$ and send $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{X})$ to all ideal components as coming from $\mathsf{X}$. If $\mathsf{X}$ is corrupted, then send $(\mathsf{globkeys}, sid, \boldsymbol{\Delta}_\mathsf{X})$ to $\mathcal{A}$. If $\Pi_\mathsf{Alg}$ is AP-secure, then there is no $\boldsymbol{\Delta}_\mathsf{P}$.
- On input $(\mathsf{compute}, sid, inputs_\mathsf{X})$ from $\mathsf{X}$ to $\mathcal{M}_\mathsf{X}$, it proceeds as follows:
  1. If $\mathsf{Alg}$ uses correlated randomness, then send $(\mathsf{crreq}, sid)$ to $\mathcal{F}_\mathsf{Alg}^\mathsf{offl}$. Get back $\mathbf{R}_\mathsf{X}$.
  2. Execute steps of the algorithm $\mathsf{Alg}$ on $inputs_\mathsf{X}$ and $\mathbf{R}_\mathsf{X}$, adding to the intermediate values $interm_\mathsf{X}$. In particular:
     - In order to invoke a subroutine $\mathsf{sub}$: come up with a session identifier $sid'$ in a deterministic manner, e.g. adding the name of the invoked subroutine and the line in $\mathsf{Alg}$ containing that invocation to $sid$. Send $(\mathsf{crreq}, sid', args_\mathsf{X})$ to $\mathcal{F}_\mathsf{sub}$, where $args_\mathsf{X}$ contains the public, and shares of the private arguments to $\mathsf{sub}$. Get back $(\mathsf{result}, sid', res_\mathsf{X})$ and add it to $interm_\mathsf{X}$.
     - In order to rearrange data (e.g. rotate vectors) or perform local computations: do them.
     - In order to perform linear computations on secret-shared values (including the sharing of constants): perform them on local shares, including the MAC shares.
     - In order to send a message $M$ (only in Alg. 10): send $(\mathsf{msg}, sid, M)$ to $\mathcal{M}_{\overline{\mathsf{X}}}$, where $\{\mathsf{X}, \overline{\mathsf{X}}\} = \{\mathsf{P}, \mathsf{S}\}$.
     - In order to receive a message $M$: receive $(\mathsf{msg}, sid, M)$ from $\mathcal{M}_{\overline{\mathsf{X}}}$.
  3. The execution has computed $res_\mathsf{X}$. Send $(\mathsf{result}, sid, res_\mathsf{X})$ back to $\mathsf{X}$.

**Protocol 3:** Generic two-party protocol from the description of an algorithm "$\mathsf{Alg}$": $\Pi_\mathsf{Alg}$

### 4.5 Pre-existing functionalities

In Sec. 3.2, we stated that we make use of certain two-party computation protocols from the literature, the description of which we do not give in this paper. We still need to give the ideal functionalities corresponding to these protocols, in order to use the UC framework to give the security proofs of the protocols that we do describe. These protocols were the following:

- Generation of a public random value modulo $q$: $\mathsf{pubrand}()$. The corresponding ideal functionality $\mathcal{F}_\mathsf{pubrand}$ is given in Func. 7.
- Generation of a random bit modulo $q$: $\mathsf{randbit}()$. The functionality $\mathcal{F}_\mathsf{randbit}$ is an instance of Func. 4, where the function $f$ takes no inputs and randomly returns either 0 or 1. The secret-sharing is done modulo $q$.
- Bit conversion from mod-2 to mod-$q$: $\mathsf{bitconv}(\langle\!\langle b \rangle\!\rangle^{(2)})$. The functionality $\mathcal{F}_\mathsf{bitconv}$ is an instance of Func. 4, where the function $f$ is the identity function, inputs are shared modulo 2 and outputs are shared modulo $q$.
- If-then-else. The functionality $\mathcal{F}_\mathsf{ite}$ is an instance of Func. 4, where the function $f$ takes three arguments: a bit $b$ and two values (or equal-length vectors

- On input (corrupt, $sid$, X) from $\mathcal{S}$ before any other commands, mark that X is corrupted and send (corrupted, $sid$) to X.
- On input (generate, $sid$) from both P and S: sample a value $v$ according to the distribution $\mathcal{D}$. Let $v_{\mathsf{P}} \leftarrow v$ and $v_{\mathsf{S}} \leftarrow v$. Send (result, $sid, v$) to $\mathcal{S}$. If party X is corrupted, then receive (update, $sid, v_{\mathsf{X}}$) from $\mathcal{S}$, otherwise receive (proceed, $sid$) from $\mathcal{S}$. Send (result, $sid, v_{\mathsf{P}}$) to P and (result, $sid, v_{\mathsf{S}}$) to S.

**Functionality 7:** Public randomness generation from the distribution $\mathcal{D}$: $\mathcal{F}^{\mathcal{D}}_{\mathsf{pubrand}}$

of values) $x$ and $y$, all secret-shared with the same modulus. The function $f$ returns $b \cdot (x - y) + y$.

### 4.6 Randomness generation

Alg. 4, parametrized with a small prime $Q$, gives an algorithm to generate a secret-shared random value between 0 and $Q-1$. It corresponds to the protocol $\Pi_{\mathsf{gen\_small}}$, which is an instance of Prot. 3. The protocol securely realizes the ideal functionality $\mathcal{F}_{\mathsf{gen\_small}}$ that is an instance of Func. 4, where the function $f$ takes no arguments and returns a random value between 0 and $Q-1$; the output is secret-shared modulo $q$. The protocol $\Pi_{\mathsf{gen\_small}}$ is realized by a system consisting of $\mathcal{M}_{\mathsf{P}}$, $\mathcal{M}_{\mathsf{S}}$, $\mathcal{F}_{\mathsf{pubrand}}$, and $\mathcal{F}^{\mathrm{offl}}_{\mathsf{gen\_small}}$ (an instance of Func. 6).

In order to show that $\Pi_{\mathsf{gen\_small}}$ is at least secure as $\mathcal{F}_{\mathsf{gen\_small}}$, we have to construct a suitable simulator. As always, we present a construction for corrupted P and another one for corrupted S.

**Simulator for corrupted phone.** During initialization, the simulator learns Phone's MAC keys via the message (globkeys, $\boldsymbol{\Delta}_{\mathsf{P}}$) from $\mathcal{F}_{\mathsf{gen\_small}}$.

During computation, the simulator gets notified of the start of the computation from the ideal functionality (normally, it would get the input share of P at this point, but there are no inputs to this computation). The simulator notifies $\mathcal{A}$ of that start (via a message presumably coming from $\mathcal{M}_{\mathsf{P}}$, telling $\mathcal{A}$ that $\mathcal{M}_{\mathsf{P}}$ received the compute-command from P). The simulator asks $\mathcal{F}_{\mathsf{gen\_small}}$ to proceed (there are not inputs to update), and receives the output $\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$. The simulator generates a random value $r \in \mathbb{Z}_Q$ and a random length-$Q$ vector $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$, such that its value at position $r$ is equal to $\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$. The simulator sends (cr, $sid$, $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$) (as if coming from $\mathcal{F}^{\mathrm{offl}}_{\mathsf{gen\_small}}$) to $\mathcal{A}$ and gets back (update, $sid$, $\langle\!\langle \mathbf{t}' \rangle\!\rangle_{\mathsf{P}}$).

The simulator expects the message (generate, $sid$) from $\mathcal{A}$ (as command for $\mathcal{M}_{\mathsf{P}}$ to send this to $\mathcal{F}_{\mathsf{pubrand}}$). The simulator (acting as $\mathcal{F}_{\mathsf{pubrand}}$) tells $\mathcal{A}$ that the result is $r$; the adversary $\mathcal{A}$ may update this to $r'$.

The adversary now tells $\mathcal{F}_{\mathsf{gen\_small}}$ to proceed and, moreover, update the output of P to $\langle\!\langle t'_{r'} \rangle\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** The protocol is symmetric wrt. the roles of Phone and Server. The simulator will be the same, with $\mathsf{P}$ and $\mathsf{S}$ swapped.

### 4.7 Key generation

Alg. 5 gives the algorithm for ML-DSA private key generation, such that the result is secret-shared. The algorithm realizes the ideal functionality $\mathcal{F}_{\mathsf{keygen}}$ that is a specialization of Func. 4, where the function $f$ receives no inputs and returns a fresh private key of ML-DSA. We define the protocol $\Pi_{\mathsf{keygen}}$ an instance of Prot. 3, corresponding to Alg. 5.

The real system realizing $\Pi_{\mathsf{keygen}}$ consists of the machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, and the ideal functionality $\mathcal{F}_{\mathsf{gen\_small}}$. Let us construct simulators for corrupted $\mathsf{P}$ and for corrupted $\mathsf{S}$.

**Simulator for corrupted phone.** During initialization, the simulator learns Phone's MAC keys via the message $(\mathsf{globkeys}, \boldsymbol{\Delta}_{\mathsf{P}})$ from $\mathcal{F}_{\mathsf{keygen}}$.

During computation, the simulator receives $\mathbf{A}$ from $\mathcal{F}_{\mathsf{keygen}}$, indicating the start of the computation. The simulator notified $\mathcal{A}$ and sends $\mathbf{A}$ to it (to $\mathcal{A}$, the origin of the message appears to be $\mathcal{M}_{\mathsf{P}}$). The simulator asks $\mathcal{F}_{\mathsf{keygen}}$ to proceed; it receives the shares $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{P}}$, and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$.

The simulator turns $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{P}}$ into the shares of the random values that are presumably generated by $\mathcal{F}_{\mathsf{gen\_small}}$. Let $\langle\!\langle r_{i,t} \rangle\!\rangle_{\mathsf{P}}$ be one of the coefficients of one of the polynomials in either $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{P}}$ or $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{P}}$. If $\eta = 2$, then define $\langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}} \leftarrow \langle\!\langle r_{i,t} \rangle\!\rangle_{\mathsf{P}} + \langle\!\langle 2 \rangle\!\rangle_{\mathsf{P}}$, where $\langle\!\langle 2 \rangle\!\rangle$ denotes the classification of the constant 2. If $\eta = 4$ then let $\langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}}$ be a random element of $\mathbb{Z}_q$ and define $\langle\!\langle r''_{i,t} \rangle\!\rangle_{\mathsf{P}} \leftarrow \langle\!\langle r_{i,t} \rangle\!\rangle_{\mathsf{P}} - 3 \cdot \langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}} + \langle\!\langle 4 \rangle\!\rangle_{\mathsf{P}}$.

The simulator waits for $\mathcal{A}$'s command (to $\mathcal{M}_{\mathsf{P}}$; supposed to be passed to $\mathcal{F}_{\mathsf{gen\_small}}$) to start some session of $\mathcal{F}_{\mathsf{gen\_small}}$; the session identifier $sid'$ used in the activation indicates, which position of $\mathbf{s}_1$ or $\mathbf{s}_2$ this invocation corresponds to (and in case of $\eta = 4$: whether it corresponds to upper or lower half of the private value). The simulator sends back $\langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}}$ or $\langle\!\langle r''_{i,t} \rangle\!\rangle_{\mathsf{P}}$, according to the position that $\mathcal{A}$ chose. The adversary $\mathcal{A}$ has to send the activations for all positions in $\mathbf{s}_1$ and $\mathbf{s}_2$. The adversary can also send updates to $\langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle r''_{i,t} \rangle\!\rangle_{\mathsf{P}}$; the simulator records these.

Once the generations of all small random values have been completed, the simulator updates $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{P}}$, recomputing them with the updated $\langle\!\langle r'_{i,t} \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle r''_{i,t} \rangle\!\rangle_{\mathsf{P}}$. The simulator tells $\mathcal{F}_{\mathsf{keygen}}$ to proceed, with updated $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** The protocol is symmetric wrt. the roles of Phone and Server. The simulator is the same as the one for corrupted Phone, with $\mathsf{P}$ and $\mathsf{S}$ swapped.

### 4.8 Zero equality checking

Alg. 6 gives the algorithm for checking whether a private value $\langle\!\langle v \rangle\!\rangle$ is equal to 0, with the precondition $v < B$, where $B$ is a rather small bound. It corresponds to a protocol $\Pi_{\mathsf{zero\_check}}$, which is an instance of Prot. 3. The protocol realizes the ideal functionality $\mathcal{F}_{\mathsf{zero\_check}}$, which is an instance of Func. 4, where the function $f$ returns 1 iff $v = 0$ (and returns 0 otherwise). The predicate $\mathbf{B}$ returns 1 if $v < B$.

The protocol $\Pi_{\mathsf{zero\_check}}$ is realized by a system consisting of machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as the functionalities $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ and $\mathcal{F}^{\mathrm{offl}}_{\mathsf{zero\_check}}$. The latter is an instance of Func. 6. Let us present the simulators to show that $\Pi_{\mathsf{zero\_check}}$ is at least as secure as $\mathcal{F}_{\mathsf{zero\_check}}$.

**Simulator for corrupted phone.** During initialization, the simulator learns Phone's MAC keys via the message $(\mathsf{globkeys}, \boldsymbol{\Delta}_{\mathsf{P}})$ from $\mathcal{F}_{\mathsf{keygen}}$.

During computation (with session ID *sid*), the simulator receives $\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$ from $\mathcal{F}_{\mathsf{zero\_check}}$, and forwards it to $\mathcal{A}$ (as if coming from $\mathcal{M}_{\mathsf{P}}$ that reports the compute-command from the environment). If the simulator does not also receive $v$, then it knows that $v < B$. The simulator asks the ideal functionality to proceed, learning $\langle\!\langle b \rangle\!\rangle_{\mathsf{P}}$, where $b$ is the output of the zero check.

The simulator generates a random $u \in \mathbb{Z}_q$, and random values $\langle\!\langle m \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$, except that the element $\langle\!\langle t_{\lfloor u/(\lfloor q/B \rfloor)\rfloor} \rangle\!\rangle_{\mathsf{P}}$ is taken to be equal to $\langle\!\langle b \rangle\!\rangle_{\mathsf{P}}$. The simulator sends $\langle\!\langle m \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$ to $\mathcal{A}$, as if coming from $\mathcal{F}^{\mathrm{offl}}_{\mathsf{zero\_check}}$. The adversary may send back adjustments to these values. Let $\langle\!\langle \mathbf{t}' \rangle\!\rangle_{\mathsf{P}}$ the adjusted value for $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$.

The simulator waits $\mathcal{A}$ to instruct $\mathcal{M}_{\mathsf{P}}$ to declassify the value $\langle\!\langle d \rangle\!\rangle_{\mathsf{P}} = \langle\!\langle m \rangle\!\rangle_{\mathsf{P}} + \lfloor q/B \rfloor \cdot \langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$. The simulator already knows $\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}$, because it is able to compute it itself. Let $\widetilde{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}$ denote the value computed by the simulator, while $\widehat{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}$ denotes the value submitted by the adversary. The simulator now computes

- ${}_{\mathsf{v}}\langle\!\langle d \rangle\!\rangle_{\mathsf{S}} \leftarrow u - {}_{\mathsf{v}}\widetilde{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}$, and
- ${}_{\mathsf{mp}}\langle\!\langle d \rangle\!\rangle_{\mathsf{S}} \leftarrow {}_{\mathsf{v}}\langle\!\langle d \rangle\!\rangle_{\mathsf{S}} \cdot \boldsymbol{\Delta}_{\mathsf{S}} - {}_{\mathsf{mp}}\widetilde{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}$,

and sends $(\mathsf{input}, sid, \widehat{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}, {}_{\mathsf{v}}\langle\!\langle d \rangle\!\rangle_{\mathsf{S}}, {}_{\mathsf{mp}}\langle\!\langle d \rangle\!\rangle_{\mathsf{S}})$ to the adversary (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$). The adversary $\mathcal{A}$ sends back an update to $\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}$. If the update is not equal to $\widetilde{\langle\!\langle d \rangle\!\rangle_{\mathsf{P}}}$ (at least for the v- and ms-parts), then the simulator tells $\mathcal{F}_{\mathsf{zero\_check}}$ to stop. Otherwise the simulator sends $(\mathsf{result?}, sid)$ to the adversary (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$) and gets back $(\mathsf{result!}, sid, u', b_{\mathsf{cnt}})$. If the boolean $b_{\mathsf{cnt}}$ is false then it tells $\mathcal{F}_{\mathsf{zero\_check}}$ to stop. If $u' \neq u$, then the simulator updates $\langle\!\langle b \rangle\!\rangle_{\mathsf{P}}$ to $\langle\!\langle t_{\lfloor u'/(\lfloor q/B \rfloor)\rfloor} \rangle\!\rangle_{\mathsf{P}}$.

At this point, the to-be-declassified value is fixed as $u'$, and the adversary is able to find the output share $\langle\!\langle b \rangle\!\rangle_{\mathsf{P}}$. The adversary can instruct $\mathcal{M}_{\mathsf{P}}$ to output something else — $\langle\!\langle b' \rangle\!\rangle_{\mathsf{P}}$ — to the environment. The simulator tells $\mathcal{F}_{\mathsf{zero\_check}}$ to proceed, with the output updated to $\langle\!\langle b' \rangle\!\rangle_{\mathsf{P}}$. Note that if $u' \neq u$, and $\mathcal{F}_{\mathsf{zero\_check}}$

is used as a component in a larger protocol (e.g. signing), then the execution of the whole protocol will halt at its next step.

If the simulator also received $v$ at the time the computation was started, then it first generates a random $m \in \mathbb{Z}_q$, and computes $u \leftarrow (m + \lfloor q/B \rfloor \cdot v) \bmod q$. It continues as before, secret-sharing $m$ and generating $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{P}}$ so that $\langle\!\langle t_{\lfloor u/(\lfloor q/B \rfloor} \rangle\!\rangle_{\mathsf{P}} = \langle\!\langle b \rangle\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** This is very similar to the simulator for corrupted phone. The only difference arises from the asymmetry of declassification.

The initialization of the components (during which the simulator learns $\boldsymbol{\Delta}_{\mathsf{S}}$) works the same way, as does the receiving of the share $\langle\!\langle v \rangle\!\rangle_{\mathsf{S}}$, the generation of the shares of correlated randomness, the updating of the correlated randomness, the computation of $\langle\!\langle d \rangle\!\rangle_{\mathsf{S}}$ and its submission to declassification. The simulator now sends $(\mathsf{input}, sid, \langle\!\langle d \rangle\!\rangle_{\mathsf{S}})$ to $\mathcal{A}$ (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$). The simulator may update it with its update-command, but in the end, the v- and mp-components must be equal to $\langle\!\langle d \rangle\!\rangle_{\mathsf{S}}$ as computed by the simulator; if they are different then the simulator tells $\mathcal{F}_{\mathsf{zero\_check}}$ to stop.

The simulator sends $(\mathsf{result}?, sid, u, \mathsf{true})$ to $\mathcal{A}$ (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$) and gets back $(\mathsf{result}, sid, u')$. If $u' \neq u$, then the simulator updates $\langle\!\langle b \rangle\!\rangle_{\mathsf{S}}$ to $\langle\!\langle t_{\lfloor u'/(\lfloor q/B \rfloor)\rfloor} \rangle\!\rangle_{\mathsf{S}}$. The adversary can instruct $\mathcal{M}_{\mathsf{S}}$ to output something else — $\langle\!\langle b' \rangle\!\rangle_{\mathsf{S}}$ — to the environment. The simulator tells $\mathcal{F}_{\mathsf{zero\_check}}$ to proceed, with the output updated to $\langle\!\langle b' \rangle\!\rangle_{\mathsf{S}}$.

### 4.9 Symmetric reshare to parts

Alg. 7 gives the algorithm for "splitting a private value into digits", parametrized with the basis $\mathbf{r} = (r_0, \ldots, r_{n-1})$. It corresponds to a protocol $\Pi_{\mathsf{symm\_rp}}$, an instance of Prot. 3, which is at least as secure as the ideal functionality $\mathcal{F}_{\mathsf{symm\_rp}}$, which is an instance of Func. 4, where

- the randomness $\hat{r}$ is a random element of $\{0, \ldots, q-1\}$;
- the function $f(v, \hat{r})$ checks whether $v \geq \hat{r}$. If so, then $f$ splits $v$ into digits. If not, then $f$ splits $v+q$ into digits. In either case, $f$ returns the characteristic vectors of these digits.

The protocol $\Pi_{\mathsf{symm\_rp}}$ is realized by a system consisting of machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as the ideal components $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$, $\mathcal{F}_{\mathsf{ite}}$, $\mathcal{F}_{\mathsf{convbit}}$ and $\mathcal{F}^{\mathrm{offl}}_{\mathsf{symm\_rp}}$. The latter is an instance of Func. 6, generating a random $s \in \mathbb{Z}_q$ and the characteristic vectors $\mathbf{d}_0, \ldots, \mathbf{d}_{n-1}$ of its digits. Let us present the simulators showing that $\Pi_{\mathsf{symm\_rp}}$ indeed securely realizes $\mathcal{F}_{\mathsf{symm\_rp}}$.

**Simulator for corrupted phone.** During initialization, the simulator learns Phone's MAC keys via the message $(\mathsf{globkeys}, \boldsymbol{\Delta}_{\mathsf{P}})$ from $\mathcal{F}_{\mathsf{keygen}}$.

During computation, the simulator receives the input $\langle\!\langle w \rangle\!\rangle_{\mathsf{P}}$ from $\mathcal{F}_{\mathsf{symm\_rp}}$, and sends it to $\mathcal{A}$. Asks $\mathcal{F}_{\mathsf{symm\_rp}}$ to proceed, and learns the output shares $\langle\!\langle \mathbf{b}_0 \rangle\!\rangle_{\mathsf{P}}, \ldots, \langle\!\langle \mathbf{b}_{n-1} \rangle\!\rangle_{\mathsf{P}}$.

The simulator is able to generate the values and random shares according to the probability distribution that $\mathcal{M}_{\mathsf{P}}$ would see them distributed. This is straightforward:

- $\langle\!\langle s \rangle\!\rangle_{\mathsf{P}}$ is a triple $({}_{\mathsf{v}}\langle\!\langle s \rangle\!\rangle_{\mathsf{P}}, {}_{\mathsf{ms}}\langle\!\langle s \rangle\!\rangle_{\mathsf{P}}, {}_{\mathsf{mp}}\langle\!\langle s \rangle\!\rangle_{\mathsf{P}})$ of random elements of $\mathbb{Z}_q$ (for the MAC shares, a vector of random elements);
- similarly, each element of the vectors $\langle\!\langle \mathbf{d}_i \rangle\!\rangle_{\mathsf{P}}$ is a triple of a random value and two vectors of random values over $\mathbb{Z}_2$;
- the additional elements of extended vectors $\langle\!\langle \mathbf{d}_i \rangle\!\rangle_{\mathsf{P}}$ are $\langle\!\langle 0 \rangle\!\rangle_{\mathsf{P}}$;
- the argument to declassification is $\langle\!\langle w \rangle\!\rangle_{\mathsf{P}} - \langle\!\langle s \rangle\!\rangle_{\mathsf{P}}$;
- $y$ is a random element of $\mathbb{Z}_q$;
- $(z_0, \ldots, z_{n-1})$ are the digits of $y$;
- $\langle\!\langle \mathbf{p}'_i \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{p}''_i \rangle\!\rangle_{\mathsf{P}}$ are rotations of $\langle\!\langle \mathbf{d}_i \rangle\!\rangle_{\mathsf{P}}$ by $z_i$ and $(z_i + 1)$ positions, respectively;
- $\langle\!\langle \mathbf{p}_0 \rangle\!\rangle_{\mathsf{P}} = \langle\!\langle \mathbf{p}'_0 \rangle\!\rangle_{\mathsf{P}}$;
- each element of the vectors $\langle\!\langle \mathbf{p}_i \rangle\!\rangle_{\mathsf{P}}$ $(i \geq 1)$ is a triple of a random value and two vectors of random values over $\mathbb{Z}_2$;
- $\langle\!\langle c_i \rangle\!\rangle_{\mathsf{P}}$ $(i \geq 1)$ is the sum (i.e. XOR, computed point-wise) of certain elements of $\langle\!\langle \mathbf{p}_i \rangle\!\rangle_{\mathsf{P}}$;
- arguments to $\mathsf{convbit}$ are sums (i.e. XORs) of certain bits in the vectors $\langle\!\langle \mathbf{p}_i \rangle\!\rangle_{\mathsf{P}}$.

The simulator does not generate these values ahead of time, but only during the simulated execution of the protocol together with the adversary $\mathcal{A}$. It starts with sending $\langle\!\langle s \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{d}_i \rangle\!\rangle_{\mathsf{P}}$ to the adversary, as if coming from $\mathcal{F}^{\mathrm{offl}}_{\mathsf{symm\_rp}}$. The adversary may adjust these values, but any adjustments to the $\mathsf{v}$- or $\mathsf{ms}$-components will lead to the simulator instructing $\mathcal{F}_{\mathsf{symm\_rp}}$ to stop execution; this instruction may come either immediately or only at the time when the adjusted value is used in $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ (for $\langle\!\langle s \rangle\!\rangle_{\mathsf{P}}$) or in $\mathcal{F}_{\mathsf{ite}}$ (for $\langle\!\langle \mathbf{d}_i \rangle\!\rangle_{\mathsf{P}}$).

The adversary will ask $\mathcal{M}_{\mathsf{P}}$ to invoke $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ in order to obtain $y$. The simulator simulates declassification identically to App. 4.8: there is again the share $\widetilde{\langle\!\langle y \rangle\!\rangle_{\mathsf{P}}}$ computed by the simulator, and the share $\widehat{\langle\!\langle y \rangle\!\rangle_{\mathsf{P}}}$ initially submitted by the adversary as-if through $\mathcal{M}_{\mathsf{P}}$ and later corrected as-if through $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$. There is the simulator computing components of $\langle\!\langle y \rangle\!\rangle_{\mathsf{S}}$ (on the basis of the value $y$ that it has chosen) and sending them to $\mathcal{A}$. In the end, the adversary may try to adjust $y$ through the $\mathsf{result}$-messages; this again leads to the stopping of the execution later, at the point where the changed digit $z_i$ is handled in the main loop.

The adversary will ask $\mathcal{M}_{\mathsf{P}}$ to invoke $\mathcal{F}_{\mathsf{ite}}$ for $n$ times in a row. If the adversary adjusts the $\mathsf{v}$- or $\mathsf{ms}$-part of the outputs of $\mathcal{F}_{\mathsf{ite}}$, then these will be caught either in the next iteration (if they change the $\mathsf{v}$- or $\mathsf{ms}$-part of the share of the carry bit), or in the invocation of $\mathcal{F}_{\mathsf{convbit}}$.

The adversary will ask $\mathcal{M}_{\mathsf{P}}$ to invoke $\mathcal{F}_{\mathsf{convbit}}$ for $\sum \mathbf{r}$ times. The simulator will give it the components of $\langle\!\langle \mathbf{b}_0 \rangle\!\rangle_{\mathsf{P}}, \ldots, \langle\!\langle \mathbf{b}_{n-1} \rangle\!\rangle_{\mathsf{P}}$ as results. The adversary

may adjust these results. The simulator tells $\mathcal{F}_{\mathsf{symm\_rp}}$ to proceed and to give the adjusted values to $\mathsf{P}$.

**Simulator for corrupted server.** The protocol is (almost) symmetric. The simulator for corrupted server works almost the same as the simulator for corrupted phone; the changes in handling asymmetric $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ are the same as in App. 4.8.

### 4.10 Finding the high bits

Alg. 8 gives the algorithm for finding the high bits of a value $\langle\!\langle w \rangle\!\rangle$. It corresponds to a protocol $\Pi_{\mathsf{high\_bits}}$, an instance of Prot. 3, which is at least as secure as the ideal functionality $\mathcal{F}_{\mathsf{high\_bits}}$, which is an instance of Func. 4, where the function $f$ computes the high bits of its argument. The real system implementing $\Pi_{\mathsf{high\_bits}}$ consists of machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as ideal components $\mathcal{F}_{\mathsf{symm\_rp}}$ and $\mathcal{F}_{\mathsf{zero\_check}}$. Let us construct the simulators showing that $\Pi_{\mathsf{high\_bits}}$ is at least as secure as $\mathcal{F}_{\mathsf{high\_bits}}$.

**Simulator for corrupted phone.** During initialization, the simulator learns Phone's MAC keys via the message $(\mathsf{globkeys}, \boldsymbol{\Delta}_{\mathsf{P}})$ from $\mathcal{F}_{\mathsf{keygen}}$.

During computation, the simulator receives the input $\langle\!\langle w \rangle\!\rangle_{\mathsf{P}}$ from $\mathcal{F}_{\mathsf{high\_bits}}$, and sends it to $\mathcal{A}$. Asks $\mathcal{F}_{\mathsf{high\_bits}}$ to proceed, and learns the output share $\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{symm\_rp}}$ with the input share $\langle\!\langle w' \rangle\!\rangle_{\mathsf{P}}$, where $\langle\!\langle w' \rangle\!\rangle_{\mathsf{P}} = \langle\!\langle w \rangle\!\rangle_{\mathsf{P}} + \langle\!\langle \alpha/2 - 1 \rangle\!\rangle_{\mathsf{P}}$. The adversary can change that input before $\mathcal{F}_{\mathsf{symm\_rp}}$ begins. If the adversary changes either the $\mathsf{v}$-component or the $\mathsf{ms}$-component of $\langle\!\langle w' \rangle\!\rangle_{\mathsf{P}}$, then the simulator tells $\mathcal{F}_{\mathsf{high\_bits}}$ to stop.

The simulator generates random vectors $\langle\!\langle \mathbf{b}_0 \rangle\!\rangle_{\mathsf{P}}, \ldots, \langle\!\langle \mathbf{b}_4 \rangle\!\rangle_{\mathsf{P}}$ and sends these to the adversary as the output shares from $\mathcal{F}_{\mathsf{symm\_rp}}$. It also computes $\langle\!\langle g \rangle\!\rangle_{\mathsf{P}} \leftarrow \langle\!\langle 5 \rangle\!\rangle_{\mathsf{P}} - \sum_{i=0}^{3} \langle\!\langle b_{i,0} \rangle\!\rangle_{\mathsf{P}} - \sum_{i=s+1}^{2s} \langle\!\langle b_{4,i} \rangle\!\rangle_{\mathsf{P}}$. The adversary may change $\langle\!\langle \mathbf{b}_0 \rangle\!\rangle_{\mathsf{P}}, \ldots, \langle\!\langle \mathbf{b}_4 \rangle\!\rangle_{\mathsf{P}}$.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{zero\_check}}$ with the input share $\langle\!\langle g \rangle\!\rangle_{\mathsf{P}}$. The adversary may change that input before $\mathcal{F}_{\mathsf{zero\_check}}$ begins. But if its $\mathsf{v}$- or $\mathsf{ms}$-components differ from the values that the simulator has previously computed, then the simulator tells $\mathcal{F}_{\mathsf{high\_bits}}$ to stop.

The simulator computes $\langle\!\langle c \rangle\!\rangle_{\mathsf{P}} \leftarrow \sum_{i=0}^{s-1} i \cdot \langle\!\langle b_{4,i} \rangle\!\rangle_{\mathsf{P}} + \sum_{i=s}^{2s} (i-s) \cdot \langle\!\langle b_{4,i} \rangle\!\rangle_{\mathsf{P}} - \langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$. The simulator sends $\langle\!\langle c \rangle\!\rangle_{\mathsf{P}}$ to the adversary as the output share from $\mathcal{F}_{\mathsf{zero\_check}}$. The adversary may now give an adjusted value of $\langle\!\langle v \rangle\!\rangle_{\mathsf{P}}$ to the simulator. The simulator forwards this to $\mathcal{F}_{\mathsf{high\_bits}}$ and asks it to proceed.

**Simulator for corrupted server.** The protocol is symmetric wrt. the roles of Phone and Server. The simulator is the same as the one for corrupted Phone, with $\mathsf{P}$ and $\mathsf{S}$ swapped.

### 4.11 Commitment computation

Alg. 9 gives the algorithm for creating the secret $\langle\!\langle \mathbf{y} \rangle\!\rangle$ and the corresponding FSwA commitment $\langle\!\langle \mathbf{w}^H \rangle\!\rangle$. We consider it as a template for the protocol $\Pi_{\mathsf{gcom}}$ (an instance of Prot. 3). The inputs and outputs of the protocol $\Pi_{\mathsf{gcom}}$ are the same as those of the ideal functionality $\mathcal{F}_{\mathsf{gcom}}$. This functionality is an instance of Func. 4, where the function $f$ receives the public matrix $\mathbf{A}$ as an input and returns a random vector $\mathbf{y}$ and $\mathbf{w}^H = \mathsf{HighBits}(\mathbf{A} \cdot \mathbf{y})$.

The protocol $\Pi_{\mathsf{gcom}}$ is realized by a system consisting of the machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as the ideal functionalities $\mathcal{F}_{\mathsf{high\_bits}}$ and $\mathcal{F}_{\mathsf{randbit}}$. Let us present the simulators showing that $\Pi_{\mathsf{gcom}}$ is at least as secure as $\mathcal{F}_{\mathsf{gcom}}$.

**Simulator for corrupted phone.** During initialization, the simulator learns the MAC keys $\boldsymbol{\Delta}_{\mathsf{P}}$ of the phone via the message $(\mathsf{globkeys}, \boldsymbol{\Delta}_{\mathsf{P}})$ from $\mathcal{F}_{\mathsf{keygen}}$.

During computation, the simulator receives the input $\mathbf{A}$ from $\mathcal{F}_{\mathsf{gcom}}$, and sends it to $\mathcal{A}$. Asks $\mathcal{F}_{\mathsf{gcom}}$ to proceed and learns the output shares $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{P}}$.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computations of $\mathcal{F}_{\mathsf{randbit}}$. The simulator comes up with random shares $\langle\!\langle b_{i,t,u} \rangle\!\rangle_{\mathsf{P}}$ for $i \in [\ell]$, $t \in [256]$, and $u \in [(\log \gamma_1) + 1]$, with the condition that the relevant linear combinations of these shares were equal to the coefficients of the elements of $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{P}}$. Concretely, for each $i \in [\ell]$ and $t \in [256]$, the simulator

- randomly generates $\langle\!\langle b_{i,t,u} \rangle\!\rangle_{\mathsf{P}}$ for $u \in \{1, \ldots, \log \gamma_1\}$,
- defines $\langle\!\langle b_{i,t,0} \rangle\!\rangle_{\mathsf{P}} \leftarrow \mathsf{cff}_t(\langle\!\langle y_i \rangle\!\rangle_{\mathsf{P}}) - \sum_{u=1}^{\log \gamma_1} 2^u \cdot \langle\!\langle b_{i,t,u} \rangle\!\rangle_{\mathsf{P}}$,

where $\mathsf{cff}_t(p)$ denotes the coefficient of $X^t$ in the polynomial $p$.

The simulator computes $\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{P}} \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{P}}$. Also, the simulator sends all $\langle\!\langle b_{i,t,u} \rangle\!\rangle_{\mathsf{P}}$ to $\mathcal{A}$, as output shares from the computations performed by $\mathcal{F}_{\mathsf{randbit}}$. The adversary may adjust these shares, and the simulator will update $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{P}}$ accordingly.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computations of $\mathcal{F}_{\mathsf{high\_bits}}$ for each coefficient of each element of $\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{P}}$. The adversary may adjust these inputs by communicating with (presumably) $\mathcal{F}_{\mathsf{high\_bits}}$. But if adversary's adjustments either to $\langle\!\langle w_{i,t} \rangle\!\rangle_{\mathsf{P}}$ or (earlier) to $\langle\!\langle b_{i,t,u} \rangle\!\rangle_{\mathsf{P}}$ cause the $\mathsf{v}$- or $\mathsf{ms}$-components of $\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{P}}$ to change from what the simulator originally computed, then the simulator will tell $\mathcal{F}_{\mathsf{gcom}}$ to stop.

The simulator will send $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{P}}$ to the adversary as the output shares of computations of $\mathcal{F}_{\mathsf{high\_bits}}$. The adversary may adjust these; the simulator will send adjusted $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{P}}$ (and also $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{P}}$) back to $\mathcal{F}_{\mathsf{gcom}}$ and tell it to proceed.

**Simulator for corrupted server.** The protocol is symmetric wrt. the roles of Phone and Server.

The simulator is the same as the one for corrupted Phone, with $\mathsf{P}$ and $\mathsf{S}$ swapped.

- On input $(\mathsf{corrupt}, sid, \mathsf{X})$ from $\mathcal{S}$ before the initialization, mark that $\mathsf{X}$ is corrupted and send $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$.
- On input $(\mathsf{init}, sid)$ from $\mathsf{P}$ and $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{S})$ from $\mathsf{S}$: Store $\boldsymbol{\Delta}_\mathsf{S}$. If $\mathsf{S}$ is corrupted, then send $(\mathsf{globkeys}, \boldsymbol{\Delta}_\mathsf{S})$ to the adversary.
- On input $(\mathsf{compute}, sid, \langle\!|v|\!\rangle_\mathsf{P})$ from $\mathsf{P}$ and $(\mathsf{compute}, sid, \langle\!|v|\!\rangle_\mathsf{S})$ from $\mathsf{S}$, do the following:
  - If a party $\mathsf{X}$ is corrupted, send $\langle\!|v|\!\rangle_\mathsf{X}$ to the adversary, and receive back an adjusted version of $\langle\!|v|\!\rangle_\mathsf{X}$.
  - Recover $v$ from the shares; check the MACs. If $\mathsf{S}$ is honest, but the MACs does not verify, then stop.
  - Pick random $v_\mathsf{P} \in \mathbb{Z}_q$ and put $v_\mathsf{S} \leftarrow (v - v_\mathsf{P}) \bmod q$.
  - Let $\mathbf{x}_\mathsf{P} \leftarrow \mathsf{split}(v_\mathsf{P})$ and $\mathbf{x}_\mathsf{S} \leftarrow \mathsf{split}(v_\mathsf{S})$. Let $\mathbf{r} \leftarrow \mathbf{x}_\mathsf{P} \cdot \boldsymbol{\Delta}_\mathsf{S}^{(Q)}$. Secret-share $\mathbf{r}$ as $[\![\mathbf{r}]\!]^{(Q)}$.
  - Define $\langle\!|\mathbf{p}|\!\rangle_\mathsf{P} \leftarrow (\mathbf{x}_\mathsf{P}, [\![\mathbf{r}]\!]_\mathsf{P})$ and $\langle\!|\mathbf{p}|\!\rangle_\mathsf{S} \leftarrow (\mathbf{x}_\mathsf{S}, [\![\mathbf{r}]\!]_\mathsf{S})$.
  - If $\mathsf{X}$ is corrupted, then send $\langle\!|\mathbf{p}|\!\rangle_\mathsf{X}$ to $\mathcal{S}$. Wait for the adversary's permission to proceed; also get updated $\langle\!|\mathbf{p}|\!\rangle_\mathsf{X}$.
  - Send $(\mathsf{result}, sid, \langle\!|\mathbf{p}|\!\rangle_\mathsf{S})$ to $\mathsf{S}$ and $(\mathsf{result}, sid, \langle\!|\mathbf{p}|\!\rangle_\mathsf{P})$ to $\mathsf{P}$.

**Functionality 8:** $\mathcal{F}_{\mathsf{asymm\_rp}}$: Ideal functionality for asymmetric reshare to parts

### 4.12 Asymmetric reshare to parts

Alg. 10 gives the algorithm for splitting Phone's share of a value into digits and adding MACs to these digits. Alg. 10 is informally expected to have AP-security, i.e. it offers security against a malicious Phone, but only privacy against a malicious Server. We can define the protocol $\Pi_{\mathsf{asymm\_rp}}$, an instance of Prot. 3, on the basis of Alg. 10. The protocol is realized by a system consisting of the machines $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$, as well as the ideal component $\mathcal{F}_{\mathsf{asymm\_rp}}^{\mathrm{offl}}$ for correlated randomness, which is an instance of Func. 6. Note that $\mathcal{F}_{\mathsf{asymm\_rp}}^{\mathrm{offl}}$ is an AP-secure functionality.

The ideal functionality $\mathcal{F}_{\mathsf{asymm\_rp}}$ securely realized by $\Pi_{\mathsf{asymm\_rp}}$ is given in Func. 8. We see that its output are the digits $\mathbf{x}_\mathsf{P}$ and $\mathbf{x}_\mathsf{S}$ of the shares of a random resharing of its input $v$; there are also Server's MACs on the digits held by the Phone. A notable omission in Func. 8 is the corrupted Server's ability to define the algorithm $\mathfrak{A}$ running on the input. Indeed, as the server is actually passive in Alg. 10, it does not get a chance to change the output so precisely; it can only change its own output share.

In order to show that $\Pi_{\mathsf{asymm\_rp}}$ is at least as secure as $\mathcal{F}_{\mathsf{asymm\_rp}}$, let us present the simulators.

**Simulator for corrupted phone.** During the initialization, the simulator does not get any information, because there is no $\boldsymbol{\Delta}_\mathsf{P}$. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_\mathsf{P} = ([\![v]\!]_\mathsf{P}, [\![V]\!]_\mathsf{P})$ from $\mathcal{F}_{\mathsf{asymm\_rp}}$ and forwards it to the adversary (as if $\mathcal{M}_\mathsf{P}$ reporting to $\mathcal{A}$ about the start of the computation). The simulator asks the ideal functionality to continue and learns the output $\langle\!|\mathbf{p}|\!\rangle_\mathsf{P} = (\mathbf{x}_\mathsf{P}, [\![\mathbf{r}]\!]_\mathsf{P})$.

The simulator computes $m \leftarrow \mathsf{split}^{-1}(\mathbf{x}_\mathsf{P})$. It generates a random $[\![M]\!]_\mathsf{P}^{(q)}$. It sends $m$, $[\![M]\!]_\mathsf{P}$ and $[\![\mathbf{r}]\!]_\mathsf{P}$ to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{asymm\_rp}}^{\mathrm{offl}}$. The simulator waits for the adversary to tell $\mathcal{M}_\mathsf{P}$ that it should send the message $(w, M_w)$ to $\mathcal{M}_\mathsf{S}$. The simulator verifies that $w = [\![v]\!]_\mathsf{P} - m$ and $M_w = [\![V]\!]_\mathsf{P} - [\![M]\!]_\mathsf{P}$. If not, then the simulator tells $\mathcal{F}_{\mathsf{asymm\_rp}}$ to stop. Otherwise, the simulator receives from $\mathcal{A}$ the values that $\mathcal{M}_\mathsf{P}$ should output to $\mathsf{P}$; the simulator tells $\mathcal{F}_{\mathsf{asymm\_rp}}$ to update $\mathsf{P}$'s outputs with these values and to proceed.

**Simulator for corrupted server.** During the initialization, the simulator learns $\boldsymbol{\Delta}_\mathsf{S}$ and forwards this to $\mathcal{A}$. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_\mathsf{S} = ([\![v]\!]_\mathsf{S}, [\![V]\!]_\mathsf{S})$ from $\mathcal{F}_{\mathsf{asymm\_rp}}$ and forwards it to the adversary. The simulator asks the ideal functionality to continue and gets the output $\langle\!|\mathbf{p}|\!\rangle_\mathsf{S} = (\mathbf{x}_\mathsf{S}, [\![\mathbf{r}]\!]_\mathsf{S})$. The simulator computes $v_\mathsf{S} = \mathsf{split}^{-1}(\mathbf{x}_\mathsf{S})$ and $w \leftarrow v_\mathsf{S} - [\![v]\!]_\mathsf{S}$. The simulator generates a random $[\![M]\!]_\mathsf{S} \in \mathbb{Z}_q$ and sends it and $[\![\mathbf{r}]\!]_\mathsf{S}$ to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{asymm\_rp}}^{\mathrm{offl}}$. The simulator tells $\mathcal{A}$ that $\mathcal{M}_\mathsf{S}$ received the message $w, M_w$ from $\mathcal{M}_\mathsf{P}$, where $M_w = w \cdot \boldsymbol{\Delta}_\mathsf{S} - ([\![V]\!]_\mathsf{S} - [\![M]\!]_\mathsf{S})$. The simulator then expects $\mathcal{A}$ to tell it, what $\mathcal{M}_\mathsf{S}$ should output to the environment; the simulator tells $\mathcal{F}_{\mathsf{asymm\_rp}}$ to update $\mathsf{S}$'s outputs with these values and to proceed.

### 4.13 Characteristic vector

Alg. 11 gives the algorithm for computing the characteristic vector of a value shared modulo $Q$, where the resulting bits are shared modulo $N$. We can define an AP-secure protocol $\Pi_{\mathsf{ch\_vec}}$ as an instance of Prot. 3, implementing Alg. 11. The protocol $\Pi_{\mathsf{ch\_vec}}$ is realized by a system consisting of machines $\mathcal{M}_\mathsf{P}$ and $\mathcal{M}_\mathsf{S}$, as well as the ideal components $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AP}}$ and $\mathcal{F}_{\mathsf{ch\_vec}}^{\mathrm{offl}}$, where the latter is an instance of Func. 6. Let us give the simulators showing that $\Pi_{\mathsf{ch\_vec}}$ is a secure implementation of $\mathcal{F}_{\mathsf{ch\_vec}}$, where the latter is an instance of Func. 5, where the function $f$ computes the characteristic vector of its input, the inputs to the functionality are secret-shared modulo $Q$, and the outputs are shared modulo $N$.

**Simulator for corrupted phone.** During the initialization, the simulator notifies $\mathcal{A}$ of the initialization. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_\mathsf{P}^{(Q)}$ from $\mathcal{F}_{\mathsf{ch\_vec}}$ and forwards it to $\mathcal{A}$. The simulator asks the ideal functionality to continue and receives the output $\langle\!|\mathbf{b}|\!\rangle_\mathsf{P}^{(N)}$.

The simulator generates a random $\langle\!|r|\!\rangle_\mathsf{P}^{(Q)}$. It also generates a random $f \in \mathbb{Z}_Q$ and defines $\langle\!|d_i|\!\rangle_\mathsf{P} \leftarrow \langle\!|b_{(i+f) \bmod Q}|\!\rangle_\mathsf{P}$ for all $i \in \mathbb{Z}_Q$. The simulator sends $\langle\!|r|\!\rangle_\mathsf{P}^{(Q)}$ and $\langle\!|\mathbf{d}|\!\rangle_\mathsf{P}$ to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{ch\_vec}}^{\mathrm{offl}}$. The adversary may respond with updates to $\langle\!|r|\!\rangle_\mathsf{P}^{(Q)}$ and $\langle\!|\mathbf{d}|\!\rangle_\mathsf{P}$; if there are updates to $\langle\!|r|\!\rangle_\mathsf{P}^{(Q)}$ then the simulator tells $\mathcal{F}_{\mathsf{ch\_vec}}$ to stop.

The simulator computes $\langle\!|f|\!\rangle_\mathsf{P} \leftarrow \langle\!|v|\!\rangle_\mathsf{P} - \langle\!|r|\!\rangle_\mathsf{P}$ and $[\![f]\!]_\mathsf{S} \leftarrow f - [\![f]\!]_\mathsf{P}$, and it expects $\mathcal{A}$ to command $\mathcal{M}_\mathsf{P}$ to tell $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AP}}$ to declassify $\langle\!|f'|\!\rangle_\mathsf{P}$. The simulator

sends $(\mathsf{input}, sid, \langle\!|f'|\!\rangle_{\mathsf{P}}, [\![f]\!]_{\mathsf{S}})$ to $\mathcal{A}$ (as if coming from $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$) and expects back an update to $\langle\!|f'|\!\rangle_{\mathsf{P}}$. If the updated share is different from $\langle\!|f|\!\rangle_{\mathsf{P}}$, then the simulator tells $\mathcal{F}_{\mathsf{ch\_vec}}$ to stop. Otherwise, the simulator tells $\mathcal{A}$ that Server's MAC check passed; in response, $\mathcal{A}$ may update the declassified value $f$ to $\hat{f}$. The simulator tells $\mathcal{F}_{\mathsf{ch\_vec}}$ to proceed, with the final result to $\mathsf{P}$ being updated to the rotation of (updated) $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{P}}$ by $\hat{f}$ positions.

**Simulator for corrupted server.** During the initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$ from $\mathcal{F}_{\mathsf{ch\_vec}}$ and forwards it to $\mathcal{A}$. At the start of the execution with session identifier $sid$, the simulator receives $\langle\!|v|\!\rangle_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. The simulator also receives $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{S}}$ from the ideal functionality.

The simulator picks a random $f \in \mathbb{Z}_Q$. It generates a random $r \in \mathbb{Z}_Q$ and secret-shares it as $\langle\!|r|\!\rangle$ (uses $\boldsymbol{\Delta}_{\mathsf{S}}$) to compute the MAC. Let $\mathbf{d}$ be the characteristic vector of $r$. The simulator secret-shares $\mathbf{d}$ as $\langle\!|\mathbf{d}|\!\rangle$ with the additional constraint that $\langle\!|d_i|\!\rangle_{\mathsf{S}} = \langle\!|b_{(i+f) \bmod Q}|\!\rangle_{\mathsf{S}}$ for each $i \in \mathbb{Z}_Q$. The simulator sends $\langle\!|r|\!\rangle_{\mathsf{S}}$ and $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{S}}$ to the adversary, as if coming from $\mathcal{F}^{\mathrm{offl}}_{\mathsf{ch\_vec}}$.

The simulator waits for $\mathcal{A}$ to instruct $\mathcal{M}_{\mathsf{S}}$ to declassify $\langle\!|g|\!\rangle_{\mathsf{S}}$. The simulator sends $(\mathsf{input}, sid, \langle\!|g|\!\rangle_{\mathsf{S}})$ to $\mathcal{A}$ (as if coming from $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$) and receives back an update for it. The simulator knows that this represents the adversarially-introduced error $\varepsilon = [\![v]\!]_{\mathsf{S}} - [\![r]\!]_{\mathsf{S}} - [\![g]\!]_{\mathsf{S}}$ in the value of $v$. The simulator defines $g \leftarrow f + \varepsilon$ and secret-shares $g$ into $\langle\!|g|\!\rangle$, with the constraint that $\langle\!|g|\!\rangle_{\mathsf{S}}$ has already been given by the adversary. This "secret-sharing under a constraint" actually means that the simulator computes

- $[\![g]\!]_{\mathsf{P}} \leftarrow g - [\![g]\!]_{\mathsf{S}}$, and
- ${}_{\mathsf{ms}}\langle\!|g|\!\rangle_{\mathsf{P}} \leftarrow [\![g]\!]_{\mathsf{P}} \cdot \boldsymbol{\Delta}_{\mathsf{S}} - {}_{\mathsf{ms}}\langle\!|g|\!\rangle_{\mathsf{S}}$.

Here all computations are modulo $Q$. Previously, $\langle\!|\mathbf{d}|\!\rangle^{(N)}$ had been computed in the same manner.

The simulator sends $(\mathsf{result?}, sid, g, \mathsf{true})$ to $\mathcal{A}$, as if coming from $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$. The adversary responds with $(\mathsf{result}, sid, g')$ to $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$, which the simulator can ignore. The simulator now creates the following description of an algorithm $\mathfrak{A}$ that would take $\langle\!|v|\!\rangle_{\mathsf{P}}$ as an argument:

- Recover $v$ from $[\![v]\!]_{\mathsf{P}}$ and $[\![v]\!]_{\mathsf{S}}$.
- Return the characteristic vector of $v + \varepsilon$.

The simulator sends the description of $\mathfrak{A}$ to the ideal functionality. The simulator expects updated $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{S}}$ from $\mathcal{A}$, which it forwards to the ideal functionality, to be outputted to $\mathsf{S}$.

### 4.14 Short overflow

Alg. 12 gives the algorithm for computing the bits that indicate whether the value $v$, secret-shared over a small modulus $Q$, is equal to a constant $r - 1$, or is greater than or equal to $r$. The returned bits may be shared over a different

modulus $N$. We can define an AP-secure protocol $\Pi_{\mathsf{short\_of}}$ as an instance of Prot. 3, implementing Alg. 12. This protocol is realized by a system consisting of the machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as the ideal component $\mathcal{F}_{\mathsf{ch\_vec}}$. The protocol securely implements the ideal functionality $\mathcal{F}_{\mathsf{short\_of}}$, which is an instance of Func. 5, where the inputs are shared modulo $Q$, the outputs are shared modulo $N$, and the function $f$ computes the previously mentioned two bits. Let us present the simulators showing that $\Pi_{\mathsf{short\_of}}$ indeed is at least as secure as $\mathcal{F}_{\mathsf{short\_of}}$.

**Simulator for corrupted phone.** During the initialization, the simulator notifies $\mathcal{A}$ of the initialization. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_{\mathsf{P}}^{(Q)}$ from $\mathcal{F}_{\mathsf{short\_of}}$ and forwards it to $\mathcal{A}$. The simulator asks $\mathcal{F}_{\mathsf{short\_of}}$ to continue and receives the output shares $\langle\!|b|\!\rangle_{\mathsf{P}}^{(N)}$ and $\langle\!|c|\!\rangle_{\mathsf{P}}^{(N)}$.

The simulator waits for $\mathcal{A}$ to instruct $\mathcal{M}_{\mathsf{P}}$ to invoke the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ with the argument $\langle\!|v|\!\rangle_{\mathsf{P}}$. If the argument received from $\mathcal{A}$ is the different from what was received from $\mathcal{F}_{\mathsf{short\_of}}$ earlier, then the simulator tells $\mathcal{F}_{\mathsf{short\_of}}$ to stop. The simulator generates a random vector $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{P}}^{(N)}$ of length $Q$, with constraints $\langle\!|d_{r-1}|\!\rangle_{\mathsf{P}} = \langle\!|b|\!\rangle_{\mathsf{P}}$ and $\sum_{i=r}^{Q-1}\langle\!|d_i|\!\rangle_{\mathsf{P}} = \langle\!|c|\!\rangle_{\mathsf{P}}$. The simulator sends $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{P}}$ to $\mathcal{A}$ as output from $\mathcal{F}_{\mathsf{ch\_vec}}$. The adversary may sends adjustments to $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{P}}$; the simulator will update $\langle\!|b|\!\rangle_{\mathsf{P}}$ and $\langle\!|c|\!\rangle_{\mathsf{P}}$ accordingly. The simulator tells $\mathcal{F}_{\mathsf{short\_of}}$ to proceed, with adjusted $\langle\!|b|\!\rangle_{\mathsf{P}}$ and $\langle\!|c|\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** During the initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. Also receives the shares $\langle\!|b|\!\rangle_{\mathsf{S}}^{(N)}$ and $\langle\!|c|\!\rangle_{\mathsf{S}}^{(N)}$ of the output.

The simulator waits for $\mathcal{A}$ to instruct $\mathcal{M}_{\mathsf{S}}$ to invoke the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ with the argument $\langle\!|v|\!\rangle_{\mathsf{S}}$; the simulator responds (as $\mathcal{F}_{\mathsf{ch\_vec}}$) that computation with $\langle\!|v|\!\rangle_{\mathsf{S}}$ has indeed started. As $\mathcal{F}_{\mathsf{ch\_vec}}$ is an instance of Func. 5, the adversary now expects to receive its share of the output from this functionality. The simulator thus generates a random $\langle\!|\mathbf{d}|\!\rangle_{\mathsf{S}}^{(N)}$ satisfying the constraints $\langle\!|d_{r-1}|\!\rangle_{\mathsf{S}} = \langle\!|b|\!\rangle_{\mathsf{S}}$ and $\sum_{i=r}^{Q-1}\langle\!|d_i|\!\rangle_{\mathsf{S}} = \langle\!|c|\!\rangle_{\mathsf{S}}$, and sends it to the adversary. The adversary now sends the description of the algorithm $\mathfrak{A}$ that it expects to be executed by $\mathcal{F}_{\mathsf{ch\_vec}}$. The algorithm $\mathfrak{A}$ expects to get $\langle\!|v|\!\rangle_{\mathsf{P}}$ as the input, and it will return a vector of booleans, which may be the characteristic vector of $v$.

The simulator will now create the description of the following algorithm $\mathcal{B}$ that takes $\langle\!|v|\!\rangle_{\mathsf{P}}$ as the input:

1. compute $\mathbf{d} \leftarrow \mathfrak{A}(\langle\!|v|\!\rangle_{\mathsf{P}})$, where $\mathbf{d} \in \mathbb{Z}_N^Q$;
2. let $b \leftarrow d_{r-1}$ and $c \leftarrow \left(\sum_{i=r}^{Q-1} d_i\right) \bmod N$;
3. return $b$ and $c$.

The simulator sends the description of $\mathfrak{B}$ to $\mathcal{F}_{\mathsf{short\_of}}$. It will get updates from $\mathcal{A}$ to $\langle\!|b|\!\rangle_{\mathsf{S}}^{(N)}$ and $\langle\!|c|\!\rangle_{\mathsf{S}}^{(N)}$ and forward these to $\mathcal{F}_{\mathsf{short\_of}}$, telling it to proceed with outputs to $\mathsf{P}$ and $\mathsf{S}$.

### 4.15 Long overflow

Alg. 13 gives the algorithm for computing the bit indicating whether the length-$d$ vector of values $\mathbf{p}$, secret-shared over a small modulus $Q$, is larger than $r^d$, when we interpret the elements of $\mathbf{p}$ as digits in basis $r$ (and the digits themselves are allowed to be larger than $r-1$). The returned bit may be shared over a different modulus $M$, and there is a "middle" modulus $N$ for the bits indicating the overflow of each single digit. We can define an AP-secure protocol $\Pi_{\mathsf{long\_of}}$ as an instance of Prot. 3, implementing Alg. 13. This protocol is realized by a system consisting of the machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as ideal components $\mathcal{F}_{\mathsf{short\_of}}$ (with input modulus $Q$ and output modulus $N$) and $\mathcal{F}_{\mathsf{ch\_vec}}$ (with input modulus $N$ and output modulus $M$). The protocol securely implements the ideal functionality $\mathcal{F}_{\mathsf{long\_of}}$, which is an instance of Func. 5, where the inputs are shared modulo $Q$, the outputs are shared modulo $M$, and the function $f$, applied to $(p_0, \ldots, p_{d-1})$, outputs whether $\sum_{i=0}^{d-1} r^i p_i$ is greater or equal to $r^d$. Let us present the simulators showing that $\Pi_{\mathsf{long\_of}}$ indeed is at least as secure as $\mathcal{F}_{\mathsf{long\_of}}$.

**Simulator for corrupted phone.** During the initialization, the simulator notifies $\mathcal{A}$ of the initialization. At the beginning of the computation, the simulator receives $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}^{(Q)}$ from $\mathcal{F}_{\mathsf{long\_of}}$ and forwards it to $\mathcal{A}$. The simulator asks $\mathcal{F}_{\mathsf{long\_of}}$ to continue and receives the output share $\langle\!|z|\!\rangle_{\mathsf{P}}^{(M)}$.The simulator generates random $\langle\!|b_i|\!\rangle_{\mathsf{P}}^{(N)}$ and $\langle\!|c_i|\!\rangle_{\mathsf{P}}^{(N)}$ for $i \in [d]$.

The simulator waits for $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{short\_of}}$ with each $\langle\!|p_i|\!\rangle_{\mathsf{P}}$. The simulator simulates $\mathcal{F}_{\mathsf{short\_of}}$ running and responds that the shares of the output are $\langle\!|b_i|\!\rangle_{\mathsf{P}}^{(N)}$ and $\langle\!|c_i|\!\rangle_{\mathsf{P}}^{(N)}$. The adversary may ask $\mathcal{F}_{\mathsf{short\_of}}$ to change either its inputs, but any such ask will result in the simulator asking $\mathcal{F}_{\mathsf{long\_of}}$ to stop. Indeed, the ideal component $\mathcal{F}_{\mathsf{short\_of}}$ of the real system would catch such changes, because it checks the MACs. The adversary may also ask $\mathcal{F}_{\mathsf{short\_of}}$ to change its outputs; the simulator will ignore these requests.

The simulator computes $\langle\!|m|\!\rangle_{\mathsf{P}} \leftarrow \sum_{i=1}^{d-1} 2^{i-1} \cdot \langle\!|b_i|\!\rangle_{\mathsf{P}} + \sum_{i=0}^{d-1} 2^i \cdot \langle\!|c_i|\!\rangle_{\mathsf{P}}$. It waits for $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ with the argument that should be equal to $\langle\!|m|\!\rangle_{\mathsf{P}}$. If it is not (after $\mathcal{A}$ has had the chance to ask $\mathcal{F}_{\mathsf{ch\_vec}}$ to change its inputs), then the simulator tells $\mathcal{F}_{\mathsf{long\_of}}$ to stop, because $\mathcal{F}_{\mathsf{ch\_vec}}$ would have caught a wrong MAC.

The simulator generates a random $\langle\!|\mathbf{k}|\!\rangle_{\mathsf{P}}^{(M)}$ of length $N$, with the constraint $\sum_{i=2^d-1}^{N-1} \langle\!|k_i|\!\rangle_{\mathsf{P}} = \langle\!|z|\!\rangle_{\mathsf{P}}$. It tells $\mathcal{A}$ that the output of $\mathcal{F}_{\mathsf{ch\_vec}}$ is $\langle\!|\mathbf{k}|\!\rangle_{\mathsf{P}}$. The adversary can submit its changes to $\langle\!|\mathbf{k}|\!\rangle_{\mathsf{P}}$. The simulator recomputes $\langle\!|z|\!\rangle_{\mathsf{P}} := \sum_{i=2^d-1}^{N-1} \langle\!|k_i|\!\rangle_{\mathsf{P}}$ and tells the adversary that this is $\mathsf{P}$'s output from the protocol. The adversary may change this $\langle\!|z|\!\rangle_{\mathsf{P}}$ as well, the simulator sends the changed value back to $\mathcal{F}_{\mathsf{long\_of}}$ and tells it to proceed with outputs.

**Simulator for corrupted server.** During the initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. At the beginning of the computation, the simulator receives $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{S}}$ from the ideal functionality and

forwards it to $\mathcal{A}$. Also receives the share $\langle\!|z|\!\rangle_{\mathsf{S}}^{(M)}$ of the output, because $\mathcal{F}_{\mathsf{long\_of}}$ is an instance of Func. 5.

The simulator waits for $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{S}}$ to start the computation of $\mathcal{F}_{\mathsf{short\_of}}$ with each $\langle\!|p_i|\!\rangle_{\mathsf{S}}$. The adversary expects to get back the shares of the output; the simulator makes up random $\langle\!|b_i|\!\rangle_{\mathsf{S}}$ and $\langle\!|c_i|\!\rangle_{\mathsf{S}}$ and sends them to $\mathcal{A}$. The adversary then sends the descriptions of algorithms $\mathfrak{A}_0, \ldots, \mathfrak{A}_{d-1}$ to the simulator (these interactions for all the $d$ invocations may be arbitrarily interleaved).

The simulator waits for $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{S}}$ to start the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ with $\langle\!|m|\!\rangle_{\mathsf{S}}^{(N)}$. If the adversary does not try to affect the computations then $\langle\!|m|\!\rangle_{\mathsf{S}} = \sum_{i=1}^{d-1} 2^{i-1} \cdot \langle\!|b_i|\!\rangle_{\mathsf{S}} + \sum_{i=0}^{d-1} 2^i \cdot \langle\!|c_i|\!\rangle_{\mathsf{S}}$, but if the adversary is performing an active attack, then it may be something else. The simulator now creates a random $\langle\!|\mathbf{k}|\!\rangle_{\mathsf{S}}^{(M)}$ of length $N$, with the constraint $\sum_{i=2^d-1}^{N-1} \langle\!|k_i|\!\rangle_{\mathsf{S}} = \langle\!|z|\!\rangle_{\mathsf{S}}$. The simulator sends $\langle\!|\mathbf{k}|\!\rangle_{\mathsf{S}}^{(M)}$ to the adversary, as if it being the output share of $\mathcal{F}_{\mathsf{ch\_vec}}$. The adversary sends the description of an algorithm $\mathfrak{B}$ to the simulator.

The simulator now constructs the following algorithm $\mathfrak{C}$ that takes $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}$ as an argument:

1. For each $i \in [d]$, let $(b_i, c_i) \leftarrow \mathfrak{A}_i(\langle\!|p_i|\!\rangle_{\mathsf{P}})$. The values $b_i$ and $c_i$ are elements of $\mathbb{Z}_N$.
2. For each $i \in [d]$, compute $\langle\!|b_i|\!\rangle_{\mathsf{P}}$ from from $b_i$ and $\langle\!|b_i|\!\rangle_{\mathsf{S}}$, using $\boldsymbol{\Delta}_{\mathsf{S}}$ to compute the MAC.
3. Similarly, compute $\langle\!|c_i|\!\rangle_{\mathsf{P}}$ for each $i \in [d]$.
4. Let $\langle\!|m|\!\rangle_{\mathsf{P}} \leftarrow \sum_{i=1}^{d-1} 2^{i-1} \cdot \langle\!|b_i|\!\rangle_{\mathsf{P}} + \sum_{i=0}^{d-1} 2^i \cdot \langle\!|c_i|\!\rangle_{\mathsf{P}}$.
5. Let $\mathbf{k} \leftarrow \mathfrak{B}(\langle\!|m|\!\rangle_{\mathsf{P}})$. The elements of the vector $\mathbf{k}$ belong to $\mathbb{Z}_M$.
6. Return $\sum_{i=2^{d-1}}^{N-1} k_i$.

The simulator sends the description of $\mathfrak{C}$ to $\mathcal{F}_{\mathsf{long\_of}}$. It may get an update from $\mathcal{A}$ to $\langle\!|z|\!\rangle_{\mathsf{S}}^{(M)}$, and will forward this to $\mathcal{F}_{\mathsf{long\_of}}$, telling it to proceed with outputs to $\mathsf{P}$ and $\mathsf{S}$.

### 4.16 Inequality check

Alg. 14 gives the algorithm to compute the bit indicating whether a secret-shared value is less than a constant. The input is shared over the modulus $q$, while the output may be shared over some different modulus $M$, and there are also moduli $Q$ and $N$ (all prime), radix $r$ and length $d$ parametrizing the algorithm. We can define an AP-secure protocol $\Pi_{\mathsf{ineq}}$ as an instance of Prot. 3, implementing Alg. 14. This protocol is realized by a system consisting of the machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, as well as ideal components $\mathcal{F}_{\mathsf{asymm\_rp}}$ (with output modulus $Q$) and $\mathcal{F}_{\mathsf{long\_of}}$ (with input modulus $Q$, "middle" modulus $N$, and output modulus $M$). The protocol securely implements the ideal functionality $\mathcal{F}_{\mathsf{ineq}}$, which is an instance of Func. 5, where the inputs are shared modulo $q$, the output bit is shared modulo $M$, and the function $f$, applied to private $v$ and public $C$, outputs whether $v < C$. Let us present the simulators showing that $\Pi_{\mathsf{ineq}}$ indeed is at least as secure as $\mathcal{F}_{\mathsf{ineq}}$.

**Simulator for corrupted phone.** During the initialization, the simulator notifies $\mathcal{A}$ of the initialization. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_{\mathsf{P}}$ and $C$ from the ideal functionality and forwards them to $\mathcal{A}$. The simulator asks $\mathcal{F}_{\mathsf{ineq}}$ to continue, and gets the output share $\langle\!|z|\!\rangle_{\mathsf{P}}^{(M)}$. The simulator waits for $\mathcal{A}$ to instruct $\mathcal{M}_{\mathsf{P}}$ to submit $\langle\!|v|\!\rangle_{\mathsf{P}}$ to $\mathcal{F}_{\mathsf{asymm\_rp}}$; the adversary may further adjust it by sending message directly to $\mathcal{F}_{\mathsf{asymm\_rp}}$. If the value finally chosen by $\mathcal{A}$ is different from $\langle\!|v|\!\rangle_{\mathsf{P}}$, then the simulator instructs $\mathcal{F}_{\mathsf{ineq}}$ to stop. Otherwise the simulator generates a random vector $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}^{(Q)}$ and sends it to the adversary, as if coming from $\mathcal{F}_{\mathsf{asymm\_rp}}$.

The simulator waits for $\mathcal{A}$'s commands to $\mathcal{M}_{\mathsf{P}}$ to submit $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}$ to $\mathcal{F}_{\mathsf{long\_of}}$ twice. The adversary is able to change the input to the two computations made by $\mathcal{F}_{\mathsf{long\_of}}$, but any change will lead to the simulator telling $\mathcal{F}_{\mathsf{ineq}}$ to stop. The simulator generates random $\langle\!|x|\!\rangle_{\mathsf{P}}^{(M)}$ and $\langle\!|y|\!\rangle_{\mathsf{P}}^{(M)}$ satisfying the constraint $\langle\!|z|\!\rangle_{\mathsf{P}} = \langle\!|1|\!\rangle_{\mathsf{P}} - \langle\!|y|\!\rangle_{\mathsf{P}} + \langle\!|x|\!\rangle_{\mathsf{p}} - \langle\!|0|\!\rangle_{\mathsf{P}}$. The simulator sends $\langle\!|x|\!\rangle_{\mathsf{P}}$ and $\langle\!|y|\!\rangle_{\mathsf{P}}$ to $\mathcal{A}$, as if coming from the two computations done by $\mathcal{F}_{\mathsf{long\_of}}$. The adversary can submit its changes to $\langle\!|x|\!\rangle_{\mathsf{P}}$ and $\langle\!|y|\!\rangle_{\mathsf{P}}$; the simulator will update $\langle\!|z|\!\rangle_{\mathsf{P}}$ accordingly. The simulator notifies $\mathcal{A}$ that the output for $\mathsf{P}$ is $\langle\!|z|\!\rangle_{\mathsf{P}}$, the adversary may change it, the simulator tells $\mathcal{F}_{\mathsf{ineq}}$ to proceed with changed $\langle\!|z|\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** During the initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. At the beginning of the computation, the simulator receives $\langle\!|v|\!\rangle_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. Also receives the share $\langle\!|z|\!\rangle_{\mathsf{S}}^{(M)}$ of the output, because $\mathcal{F}_{\mathsf{ineq}}$ is an instance of Func. 5.

Wait for $\mathcal{A}$ to instruct $\mathcal{S}$ to submit $\langle\!|v|\!\rangle_{\mathsf{S}}$ to $\mathcal{F}_{\mathsf{asymm\_rp}}$. Inform (as $\mathcal{F}_{\mathsf{asymm\_rp}}$) $\mathcal{A}$ that $\langle\!|v|\!\rangle_{\mathsf{S}}$ was submitted, allow $\mathcal{A}$ to update it. Let $\varepsilon \in \mathbb{Z}_q$ be the difference between the updated $[\![v]\!]_{\mathsf{S}}$ and the original $[\![v]\!]_{\mathsf{S}}$ received from $\mathcal{F}_{\mathsf{ineq}}$. Generate a random $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{S}}^{(Q)}$ and send it to $\mathcal{A}$ as if coming from $\mathcal{F}_{\mathsf{asymm\_rp}}$. Let $w_{\mathsf{S}} \leftarrow \mathsf{split}^{-1}([\![\mathbf{p}]\!]_{\mathsf{S}})$. Let $\langle\!|c|\!\rangle_{\mathsf{S}}^{(M)} \leftarrow \langle\!|0|\!\rangle_{\mathsf{S}}$; update $[\![c]\!]_{\mathsf{S}}$ to 1 if $w_{\mathsf{S}} \geq C$.

Wait for $\mathcal{A}$ to instruct $\mathcal{S}$ to submit $\langle\!|\mathbf{a}|\!\rangle_{\mathsf{S}}^{(Q)}$ and $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{S}}^{(Q)}$ to $\mathcal{F}_{\mathsf{long\_of}}$ for computation. The adversary expects to get the output shares $\langle\!|x|\!\rangle_{\mathsf{S}}^{(M)}$ and $\langle\!|y|\!\rangle_{\mathsf{S}}^{(M)}$. The simulator randomly generates them, such that the constraint $\langle\!|z|\!\rangle_{\mathsf{S}} = \langle\!|1|\!\rangle_{\mathsf{S}} + \langle\!|x|\!\rangle_{\mathsf{S}} - \langle\!|y|\!\rangle_{\mathsf{S}} - \langle\!|c|\!\rangle_{\mathsf{S}}$ is satisfied, and sends them to $\mathcal{A}$. The adversary sends the descriptions of algorithms $\mathfrak{A}_1$ and $\mathfrak{A}_2$ meant for the two computations of $\mathcal{F}_{\mathsf{long\_of}}$. These algorithms expect $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}$ as an input. The simulator creates the description of the following algorithm $\mathcal{B}$, expecting $\langle\!|v|\!\rangle_{\mathsf{P}}$ as an input:

1. Recover $v$ from $[\![v]\!]_{\mathsf{P}}$ and $[\![v]\!]_{\mathsf{S}}$.
2. Let $[\![\mathbf{p}]\!]_{\mathsf{P}} \leftarrow \mathsf{split}(v + \varepsilon) - [\![\mathbf{p}]\!]_{\mathsf{S}}$. I.e. $[\![\mathbf{p}]\!]_{\mathsf{P}}$ is computed so that the condition for the output of Alg. 10 holds, if we put $(v + \varepsilon) \bmod q$ in the position of $v$ there.
3. Extend $[\![\mathbf{p}]\!]_{\mathsf{P}}$ to $\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}}$ by computing $\mathsf{P}$'s MAC share from $[\![\mathbf{p}]\!]_{\mathsf{P}}$, $\boldsymbol{\Delta}_{\mathsf{S}}$ and $\mathsf{S}$'s MAC share.
4. Let $x \leftarrow \mathfrak{A}_1(\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}})$ and $y \leftarrow \mathfrak{A}_2(\langle\!|\mathbf{p}|\!\rangle_{\mathsf{P}})$. Here $x, y \in \mathbb{Z}_M$.

5. Output $(1 + x - y - c) \bmod M$.

The simulator sends the description of $\mathfrak{B}$ to $\mathcal{F}_{\mathsf{ineq}}$. It may get an update from $\mathcal{A}$ to $\langle\!|z|\!\rangle_{\mathsf{S}}^{(M)}$, and will forward this to $\mathcal{F}_{\mathsf{ineq}}$, telling it to proceed with outputs to $\mathsf{P}$ and $\mathsf{S}$.

### 4.17 Rejection check

Alg. 15 gives the algorithm for the "second half" of a signing attempt: the rejection check. It takes a number of secret-shared values and returns 1 if all these values are inside bounds. We can define an AP-secure protocol $\Pi_{\mathsf{rej\_check}}$ as an instance of Prot. 3, implementing Alg. 15. The protocol $\Pi_{\mathsf{rej\_check}}$ is implemented by a system consisting of machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, and the ideal components $\mathcal{F}_{\mathsf{ineq}}$ and $\mathcal{F}_{\mathsf{ch\_vec}}$ (with parameters named in Alg. 15). The protocol $\Pi_{\mathsf{rej\_check}}$ is at least as secure as the ideal functionality $\mathcal{F}_{\mathsf{rej\_check}}$, which is an instance of Func. 5, where the function $f$ receives $\mathbf{z}$ and $\mathbf{x}$ as private inputs, and outputs a bit showing whether they all passed the rejection check. Let us present the simulators that attest to this.

**Simulator for corrupted phone.** During the initialization, the simulator notifies $\mathcal{A}$ of the initialization. At the beginning of the computation, the simulator receives $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{P}}$ from the ideal functionality and forwards them to $\mathcal{A}$. The simulator asks $\mathcal{F}_{\mathsf{rej\_check}}$ to continue and receives the output share $\langle\!|b_0|\!\rangle_{\mathsf{P}}^{(2)}$.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{ineq}}$ for each coefficient of each element of $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{P}}$ (shifted by appropriate amount). The adversary has the chance to change these coefficients, but any change will result in the simulator asking $\mathcal{F}_{\mathsf{rej\_check}}$ to stop. The simulator generates a random vector $\langle\!|\mathbf{v}|\!\rangle_{\mathsf{P}}$ (using the modulus $M$) of the correct length $(256 \cdot (k + \ell))$, and uses the elements of this vector as Phone's output shares from all the computations of $\mathcal{F}_{\mathsf{ineq}}$. The simulator sends the vector $\langle\!|\mathbf{v}|\!\rangle_{\mathsf{P}}^{(M)}$ to $\mathcal{A}$, element by element.

The simulator computes $\langle\!|s|\!\rangle_{\mathsf{P}} \leftarrow \sum_{i=0}^{256(k+\ell)-1}(\langle\!|1|\!\rangle_{\mathsf{P}} - \langle\!|v_i|\!\rangle_{\mathsf{P}})$ and expects $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{P}}$ to start the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ with the input $\langle\!|s|\!\rangle_{\mathsf{P}}$. If the input is anything else (after the adversary has had its opportunity to change it by communicating with (presumably) $\mathcal{F}_{\mathsf{ch\_vec}}$, then the simulator tells $\mathcal{F}_{\mathsf{rej\_check}}$ to stop. Otherwise, the simulator generates a random vector $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{P}}^{(2)}$, where $\langle\!|b_0|\!\rangle_{\mathsf{P}}$ has already been defined. The simulator sends $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{P}}$ to the adversary. The adversary may make changes; the simulator tells $\mathcal{F}_{\mathsf{rej\_check}}$ to proceed, using the perhaps changed $\langle\!|b_0|\!\rangle_{\mathsf{P}}$.

**Simulator for corrupted server.** During the initialization, the simulator receives $\boldsymbol{\Delta}_{\mathsf{S}}$ from the ideal functionality and forwards it to $\mathcal{A}$. At the beginning of the computation, the simulator receives $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{S}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{S}}$ from the ideal functionality and forwards them to $\mathcal{A}$. The simulator also receives the output share $\langle\!|b_0|\!\rangle_{\mathsf{S}}$ from the adversary.

The simulator waits for $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{S}}$ to start the computation of $\mathcal{F}_{\mathsf{ineq}}$ for each coefficient of each element of $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{S}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{S}}$ (shifted by appropriate amount). The adversary expects to get the output shares of the results of each comparison. Hence the simulator generates a random $\langle\!|\mathbf{v}|\!\rangle_{\mathsf{S}}$ (of the correct length) and sends it to $\mathcal{A}$, element by element. The adversary responds by sending the algorithm descriptions of the algorithms $\mathfrak{A}_0, \ldots, \mathfrak{A}_{256(k+\ell)-1}$, each of them expecting a share $\langle\!|z_i|\!\rangle_{\mathsf{P}}$ or $\langle\!|x_i|\!\rangle_{\mathsf{P}}$ as input and producing an element of $\mathbb{Z}_M$ as output, to the simulator.

The simulator computes $\langle\!|s|\!\rangle_{\mathsf{S}} \leftarrow \sum_{i=0}^{256(k+\ell)-1}(\langle\!|1|\!\rangle_{\mathsf{S}} - \langle\!|v_i|\!\rangle_{\mathsf{S}})$ and expects $\mathcal{A}$ to ask $\mathcal{M}_{\mathsf{S}}$ to start the computation of $\mathcal{F}_{\mathsf{ch\_vec}}$ for $\langle\!|s|\!\rangle_{\mathsf{S}}$. The adversary expects to get the output share of this computation; the simulator generates a random $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{S}}^{(2)}$, where the element $\langle\!|b_0|\!\rangle_{\mathsf{S}}$ has already been received from $\mathcal{F}_{\mathsf{rej\_check}}$, and sends $\langle\!|\mathbf{b}|\!\rangle_{\mathsf{S}}$ to $\mathcal{A}$. The adversary responds by sending to the simulator the description of the algorithm $\mathfrak{B}$ that takes a share $\langle\!|s|\!\rangle_t^{(M)} extsfP$ as input and returns a vector of bits.

The simulator creates the description of the following algorithm $\mathfrak{C}$ that takes $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{P}}$ as input and outputs a bit:

1. Apply each of $\mathfrak{A}_i$ to one of the elements of $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ or $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{P}}$, shifted by the required amount (either by $\gamma_1 - \beta - 1$ or by $\gamma_2 - \beta - 1$). Let $\mathbf{v} \in \mathbb{Z}_M^{256(k+\ell)}$ be the vector of outputs from these algorithms $\mathfrak{A}_i$.
2. Let $s \leftarrow \sum_i v_i$. Compute $\langle\!|s|\!\rangle_{\mathsf{P}}$ so, that $(\langle\!|s|\!\rangle_{\mathsf{P}}, \langle\!|s|\!\rangle_{\mathsf{S}})$ will be a secret-sharing of $s$. This computation uses $\boldsymbol{\Delta}_{\mathsf{S}}$.
3. Let $\mathbf{b} \leftarrow \mathfrak{B}(\langle\!|s|\!\rangle_{\mathsf{P}})$.
4. Return $b_0$.

The simulator sends the description of $\mathfrak{C}$ to $\mathcal{F}_{\mathsf{rej\_check}}$. It may get an update from $\mathcal{A}$ to $\langle\!|b_0|\!\rangle_{\mathsf{S}}^{(2)}$, and will forward this to $\mathcal{F}_{\mathsf{rej\_check}}$, telling it to proceed with outputs to $\mathsf{P}$ and $\mathsf{S}$.

## 5 Security of the ML-DSA protocol

Protocol $\Pi_{\mathsf{ML\text{-}DSA}}$, given in Prot. 4, corresponds to Alg. 16, but it also contains the rest of the functionality implementing $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ (Func. 1) — initialization and key generation. The system implementing $\Pi_{\mathsf{ML\text{-}DSA}}$ consists of the two protocol machines $\mathcal{M}_{\mathsf{P}}$ and $\mathcal{M}_{\mathsf{S}}$, and six ideal components that we have described above: $\mathcal{F}_{\mathsf{keygen}}$, $\mathcal{F}_{\mathsf{gcom}}$, $\mathcal{F}_{\mathsf{rej\_check}}$, $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AA}}$, $\mathcal{F}_{\mathsf{declassify}}^{\mathrm{AP}}$, and $\mathcal{F}_{\mathsf{pubrand}}^{\mathcal{U}}$, where $\mathcal{U}$ is the uniform distribution over matrices $\mathbf{A}$ in the public key of ML-DSA.

To show that $\Pi_{\mathsf{ML\text{-}DSA}}$ is at least as secure as $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, we need to construct a simulator. We describe it for the case of corrupted phone, and for the case of corrupted server.

### 5.1 Simulator for corrupted phone

**Start.** In the following, the adversary $\mathcal{A}$ expects to send messages to $\mathcal{M}_{\mathsf{P}}$ and the ideal components in $\Pi_{\mathsf{ML\text{-}DSA}}$, as well as receive messages from them. All these

- On input $(\mathsf{corrupted}, sid)$ from $\mathcal{A}$ or any ideal component to $\mathcal{M}_\mathsf{X}$ before the initialization, machine $\mathcal{M}_\mathsf{X}$ sends $(\mathsf{corrupted}, sid)$ to $\mathsf{X}$ and becomes corrupted.
- On input $(\mathsf{keygen}, sid)$ from $\mathsf{X}$ to $\mathcal{M}_\mathsf{X}$, machine $\mathcal{M}_\mathsf{X}$ generates MAC keys $\boldsymbol{\Delta}_\mathsf{X}$. Sends $(\mathsf{init}, sid, \boldsymbol{\Delta}_\mathsf{X})$ to all ideal components that have to be initialized (Messages from $\mathcal{M}_\mathsf{P}$ to $\mathcal{F}_{\mathsf{rej\_check}}$ and $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$ do not include $\boldsymbol{\Delta}_\mathsf{P}$). Machine $\mathcal{M}_\mathsf{X}$ will then
  1. send $(\mathsf{generate}, sid)$ to $\mathcal{F}^{\mathcal{U}}_{\mathsf{pubrand}}$ and get back $(\mathsf{result}, sid, \mathbf{A})$;
  2. send $(\mathsf{compute}, sid)$ to $\mathcal{F}_{\mathsf{keygen}}$ and get back $(\mathsf{result}, sid, \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{X}, \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{X})$;
  3. Compute $\langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{X} \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{X} + \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{X}$;
  4. send $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{X})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ and get back $(\mathsf{result}, sid, \mathbf{t})$.

  It stores the received values and sends $(\mathsf{pkey}, sid, \mathbf{A}, \mathbf{t})$ back to $\mathsf{X}$.
- On input $(\mathsf{sign}, sid, \mu)$ from $\mathsf{P}$ to $\mathcal{M}_\mathsf{P}$, the machine $\mathcal{M}_\mathsf{P}$ works as follows, where $\mathsf{X} \equiv \mathsf{P}$:
  1. Send $(\mathsf{compute}, sid, \mathbf{A})$ to $\mathcal{F}_{\mathsf{gcom}}$ and get back $(\mathsf{result}, sid, \langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{X}, \langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{X})$
  2. Compute $\langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{X} \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{X} - \alpha \cdot \langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{X}$
  3. Send $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{w}^\mathbf{H} \rangle\!\rangle_\mathsf{X})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$ and get back $(\mathsf{result}, sid, \mathbf{w}^H)$
  4. Compute $c \leftarrow H(\mu, \mathbf{w}^H)$, $\langle\!\langle \mathbf{z} \rangle\!\rangle_\mathsf{X} \leftarrow \langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{X} + c \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{X}$, $\langle\!\langle \mathbf{x} \rangle\!\rangle_\mathsf{X} \leftarrow \langle\!\langle \mathsf{w}^L \rangle\!\rangle_\mathsf{X} - c \cdot \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{X}$
  5. Obtain $\langle\!| \mathbf{z} |\!\rangle_\mathsf{X}$ and $\langle\!| \mathbf{x} |\!\rangle_\mathsf{X}$ from $\langle\!\langle \mathbf{z} \rangle\!\rangle_\mathsf{X}$ and $\langle\!\langle \mathbf{x} \rangle\!\rangle_\mathsf{X}$ by forgetting the shares of $\mathsf{P}$'s MACs
  6. Send $(\mathsf{compute}, sid, \langle\!| \mathbf{z} |\!\rangle_\mathsf{X}, \langle\!| \mathbf{x} |\!\rangle_\mathsf{X})$ to $\mathcal{F}_{\mathsf{rej\_check}}$ and get back $(\mathsf{result}, sid, \langle\!| b |\!\rangle_\mathsf{X})$
  7. Send $(\mathsf{declassify}, sid, \langle\!| b |\!\rangle_\mathsf{X})$ to $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$ and get back $(\mathsf{result}, sid, b)$
  8. If $b = 0$, send $(\mathsf{result}, sid, \perp)$ to $\mathsf{X}$
  9. If $b = 1$, then receive $(\mathsf{ss}, sid, [\![\mathbf{z}]\!]_\mathsf{S})$ from $\mathcal{M}_\mathsf{S}$. Recover $\mathbf{z}$. Verify the signature $(\mathbf{z}, c)$ on the message $\mu$. If it verifies, send $(\mathsf{sig}, sid, \mathbf{z})$ to $\mathcal{M}_\mathsf{S}$ and $(\mathsf{signature}, sid, (\mathbf{z}, c))$ to $\mathsf{P}$. If it does not verify, then stop working.
- On input $(\mathsf{sign}, sid, \mu)$ from $\mathsf{S}$ to $\mathcal{M}_\mathsf{S}$, the machine $\mathcal{M}_\mathsf{S}$ works as follows:
  1. Perform the steps (1)–(8) above, with $\mathsf{X} \equiv \mathsf{S}$
  9. If $b = 1$, then send $(\mathsf{ss}, sid, [\![\mathbf{z}]\!]_\mathsf{S})$ to $\mathcal{M}_\mathsf{P}$. Receive $(\mathsf{sig}, sid, \mathbf{z})$ from $\mathcal{M}_\mathsf{P}$. Verify the signature $(\mathbf{z}, c)$ on the message $\mu$. If it verifies, send $(\mathsf{signature}, sid, (\mathbf{z}, c))$ to $\mathsf{S}$. If it does not verify, then stop working.

**Protocol 4:** ML-DSA protocol, $\Pi_{\mathsf{ML\text{-}DSA}}$

messages terminate or originate at the simulator *Sim* that we are constructing. The messages seemingly going from $\mathcal{M}_\mathsf{P}$ to $\mathcal{A}$ are the reports of the messages and queries that $\mathcal{M}_\mathsf{P}$ has received from other components of $\Pi_{\mathsf{ML\text{-}DSA}}$ and the environment $\mathcal{Z}$, while the messages from $\mathcal{A}$ to $\mathcal{M}_\mathsf{P}$ are commands to send a message to a certain component or $\mathcal{Z}$. The simulator *Sim* has already received from the adversary $\mathcal{A}$ the corruption query for $\mathcal{M}_\mathsf{P}$, and forwarded to $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$.

**Key generation.** On input $(\mathsf{pkey}, sid, \mathbf{A}, \mathbf{t})$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, proceed as follows. Generate the MAC keys $\boldsymbol{\Delta}_\mathsf{S}$. Notify $\mathcal{A}$ of the start of key generation; receive MAC keys $\boldsymbol{\Delta}_\mathsf{P}$ from it (as part of the $(\mathsf{init})$-queries that $\mathcal{M}_\mathsf{P}$ has to send to ideal components); send these keys back to $\mathcal{A}$ inside $(\mathsf{globkeys})$-queries from ideal components.

Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{generate}, sid)$ to $\mathcal{F}_{\mathsf{pubrand}}$. Send $(\mathsf{result}, sid, \mathbf{A})$ to $\mathcal{A}$.

Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{compute}, sid, \mathbf{A})$ to $\mathcal{F}_{\mathsf{keygen}}$. Generate random $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{P}$ and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{P}$. Compute $\langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{P} \leftarrow \mathbf{A} \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{P} + \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{P}$. Send $(\mathsf{result}, sid, \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{P}, \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{P})$ to $\mathcal{A}$.

Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{P})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$. Compute $[\![\mathbf{t}]\!]_\mathsf{S} \leftarrow \mathbf{t} - [\![\mathbf{t}]\!]_\mathsf{P}$ and send (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$) the message $(\mathsf{input}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{P}, [\![\mathbf{t}]\!]_\mathsf{S}, [\![[\![\mathbf{t}]\!]_\mathsf{S} \cdot \boldsymbol{\Delta}_\mathsf{P}]\!]_\mathsf{S})$ to $\mathcal{A}$. Receive $\mathcal{A}$'s updates to $\langle\!\langle \mathbf{t} \rangle\!\rangle_\mathsf{P}$. If $\mathcal{A}$ has changed $[\![\mathbf{t}]\!]_\mathsf{P}$ or Server's MAC on it, then tell $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to stop. Send $(\mathsf{result?}, sid)$ to $\mathcal{A}$ (as if coming from $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$), receive any updates to $\mathbf{t}$, send these to $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ and allow it to proceed.

**Signing.** On input $(\mathsf{signature}, sid, \mu, \sigma)$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, where the signature $\sigma$ is $(\mathbf{z}, c)$, proceed as follows. The simulator finds $\mathbf{x}_0 \leftarrow A \cdot \mathbf{z} - c \cdot \mathbf{t}$, $\mathbf{w}^H \leftarrow \mathbf{x}_0^H$, and $\mathbf{x} \leftarrow \mathbf{x}_0 - \alpha \cdot \mathbf{w}^H$.

The simulator randomly generates $\langle\!\langle \mathbf{z} \rangle\!\rangle_\mathsf{P}$ and $\langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{P}$.

The simulator computes

- $\langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P} \leftarrow \langle\!\langle \mathbf{z} \rangle\!\rangle_\mathsf{P} - c \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_\mathsf{P}$
- $\langle\!\langle \mathbf{w} \rangle\!\rangle_\mathsf{P} \leftarrow A \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P}$
- $\langle\!\langle \mathbf{x} \rangle\!\rangle_\mathsf{P} \leftarrow \langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{P} - c \cdot \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_\mathsf{P}$
- $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P} \leftarrow \alpha^{-1} \cdot (\langle\!\langle \mathbf{w} \rangle\!\rangle_\mathsf{P} - \langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{P})$
- $[\![\mathbf{z}]\!]_\mathsf{S} \leftarrow \mathbf{z} - [\![\mathbf{z}]\!]_\mathsf{P}$

The simulator (as $\mathcal{M}_\mathsf{P}$) notifies $\mathcal{A}$ that a signing session for $\mu$ has started. Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{compute}, sid, \mathbf{A})$ to $\mathcal{F}_{\mathsf{gcom}}$. Send $\langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P}$ and $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P}$ to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{gcom}}$.

Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$, and $\mathcal{A}$'s updates to $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P}$. Continue as in the simulation of declassification during key generation; $\mathcal{A}$ will learn $\mathbf{w}^H$.

Receive $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{compute}, sid, \langle\!|\mathbf{z}|\!\rangle_\mathsf{P}, \langle\!|\mathbf{x}|\!\rangle_\mathsf{P})$ to $\mathcal{F}_{\mathsf{rej\_check}}$. The simulator already knows the expected values of $\langle\!|\mathbf{z}|\!\rangle_\mathsf{P}$ and $\langle\!|\mathbf{x}|\!\rangle_\mathsf{P}$; if the adversary submits something else, then the simulator treats it as a failure in verifying Server's MACs and tells the ideal functionality to not output the signature. Simulator generates a random $\langle\!|b|\!\rangle_\mathsf{P}$ and sends it to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{rej\_check}}$. Following that, the simulator expects $\mathcal{A}$'s command to $\mathcal{M}_\mathsf{P}$ to send $(\mathsf{declassify}, sid, \langle\!|b|\!\rangle_\mathsf{P})$ to $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$. Perform the declassification as above; $\mathcal{A}$ learns that $b = 1$.

The simulator follows this up with $(\mathsf{ss}, sid, [\![\mathbf{z}]\!]_\mathsf{S})$, presumably from $\mathcal{M}_\mathsf{S}$ to $\mathcal{M}_\mathsf{P}$. The simulator expects $(\mathsf{sig}, sid, \mathbf{z})$ from $\mathcal{A}$ (as if $\mathcal{M}_\mathsf{P}$ sent it to $\mathcal{M}_\mathsf{S}$), and, having received this, instructs the ideal functionality to output the signature.

But if the simulator received the query $(\mathsf{signature}, sid, M, \perp)$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, then the simulator creates a random $\mathbf{w}^H$ and computes $c = H(M, \mathbf{w}^H)$. It also creates random $\langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P}$ and $\langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{P}$. It computes $\langle\!\langle \mathbf{z} \rangle\!\rangle_\mathsf{P}$, $\langle\!\langle \mathbf{w} \rangle\!\rangle_\mathsf{P}$, $\langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P}$, and $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P}$ as before, and again sends $\langle\!\langle \mathbf{y} \rangle\!\rangle_\mathsf{P}$, $\langle\!\langle \mathbf{w}^L \rangle\!\rangle_\mathsf{P}$, $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_\mathsf{P}$, and $\mathbf{w}^H$ to the adversary through the same interactions as before. It again expects to get back $\langle\!|z|\!\rangle_\mathsf{P}$ and

$\langle\!|x|\!\rangle_{\mathsf{P}}$ from the adversary, sent as inputs to $\mathcal{F}_{\mathsf{rej\_check}}$. The simulator then tells the adversary that the result of rejection checking is 0.

### 5.2 Simulator for corrupted server

**Key generation.** The key generation protocol and ideal functionality are symmetric between the Phone and the Server, except for the slight asymmetry of $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$. Similarly, the simulator learns $(\mathbf{A}, \mathbf{t})$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, generates $\boldsymbol{\Delta}_{\mathsf{P}}$, gets $\boldsymbol{\Delta}_{\mathsf{S}}$ from the adversary, generates $\langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{S}}$ and $\langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{S}}$ and sends them to $\mathcal{A}$. The simulator then expects $\mathcal{A}$ to tell $\mathcal{M}_{\mathsf{S}}$ to submit $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle'_{\mathsf{S}})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$, where $\langle\!\langle \mathbf{t} \rangle\!\rangle'_{\mathsf{S}}$ is supposed to be equal to $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{S}} = \mathbf{A} \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{S}} + \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{S}}$. The simulator, acting as $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$, sends $(\mathsf{input}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle'_{\mathsf{S}})$ back to $\mathcal{A}$. The adversary $\mathcal{A}$ returns $(\mathsf{update}, sid, \langle\!\langle \mathbf{t} \rangle\!\rangle''_{\mathsf{S}}, \mathbf{t}')$ to the simulator. If $\langle\!\langle \mathbf{t} \rangle\!\rangle''_{\mathsf{S}}$ differs from $\langle\!\langle \mathbf{t} \rangle\!\rangle_{\mathsf{S}}$ (except for the share of Server's MAC) then the simulator tells $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to stop. Otherwise the simulator tells $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to return the public key to $\mathsf{P}$, and the public key with $\mathcal{A}$'s updates $(\mathbf{A}', \mathbf{t}')$ to $\mathsf{S}$.

**Successful signing.** On input $(\mathsf{signature}, sid, \mu, (\mathbf{z}, c)))$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, proceed as follows. The simulator finds $\mathbf{x}_0 \leftarrow \mathbf{A}\cdot\mathbf{z} - c\cdot\mathbf{t}$, $\mathbf{w}^H \leftarrow \mathbf{x}_0^H$, and $\mathbf{x} \leftarrow \mathbf{x}_0 - \alpha\cdot\mathbf{w}^H$. It secret-shares $\mathbf{z}$ into $\langle\!\langle \mathbf{z} \rangle\!\rangle$, using $\boldsymbol{\Delta}_{\mathsf{S}}$ and $\boldsymbol{\Delta}_{\mathsf{P}}$ to compute the MACs. The simulator randomly generates $\langle\!\langle \mathbf{w}^L \rangle\!\rangle_{\mathsf{S}}$. The simulator computes

- $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{S}} \leftarrow \langle\!\langle \mathbf{z} \rangle\!\rangle_{\mathsf{S}} - c \cdot \langle\!\langle \mathbf{s}_1 \rangle\!\rangle_{\mathsf{S}}$
- $\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{S}} \leftarrow A \cdot \langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{S}}$
- $\langle\!\langle \mathbf{x} \rangle\!\rangle_{\mathsf{S}} \leftarrow \langle\!\langle \mathbf{w}^L \rangle\!\rangle_{\mathsf{S}} - c \cdot \langle\!\langle \mathbf{s}_2 \rangle\!\rangle_{\mathsf{S}}$
- $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{S}} \leftarrow \alpha^{-1} \cdot (\langle\!\langle \mathbf{w} \rangle\!\rangle_{\mathsf{S}} - \langle\!\langle \mathbf{w}^L \rangle\!\rangle_{\mathsf{S}})$

The simulator also computes $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{P}}$ and $\langle\!\langle \mathbf{x} \rangle\!\rangle_{\mathsf{P}}$ on the basis of $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{S}}$, $\mathbf{w}^H$, $\langle\!\langle \mathbf{x} \rangle\!\rangle_{\mathsf{S}}$, $\mathbf{x}$, $\boldsymbol{\Delta}_{\mathsf{P}}$ and $\boldsymbol{\Delta}_{\mathsf{S}}$. The simulator (as $\mathcal{M}_{\mathsf{S}}$) notifies $\mathcal{A}$ that a signing session for $\mu$ has started. Receives $\mathcal{A}$'s command to $\mathcal{M}_{\mathsf{S}}$ to send $(\mathsf{compute}, sid, \mathbf{A})$ to $\mathcal{F}_{\mathsf{gcom}}$. Sends $\langle\!\langle \mathbf{y} \rangle\!\rangle_{\mathsf{S}}$ and $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{S}}$ to $\mathcal{A}$, as if coming from $\mathcal{F}_{\mathsf{gcom}}$.

Next, the simulator receives $\mathcal{A}$'s command to $\mathcal{M}_{\mathsf{S}}$ to send the message $(\mathsf{declassify}, sid, \langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{S}})$ to $\mathcal{F}^{\mathrm{AA}}_{\mathsf{declassify}}$, and $\mathcal{A}$'s updates to $\langle\!\langle \mathbf{w}^H \rangle\!\rangle_{\mathsf{S}}$. Continue as in the simulation of declassification during key generation; $\mathcal{A}$ will learn $\mathbf{w}^H$.

The simulator expects $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{S}}$ to send to $\mathcal{F}_{\mathsf{rej\_check}}$ the message $(\mathsf{compute}, sid, \langle\!|\mathbf{z}|\!\rangle_{\mathsf{S}}, \langle\!|\mathbf{x}|\!\rangle_{\mathsf{S}})$. The simulator generates a random $\langle\!|b|\!\rangle_{\mathsf{S}}$ and sends it back to $\mathcal{A}$ (as if coming from $\mathcal{F}_{\mathsf{rej\_check}}$. The simulator then receives the description of the algorithm $\mathfrak{A}$ from $\mathcal{A}$ and applies it to $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{P}}$ that it generated while secret-sharing $\mathbf{z}$ and $\mathbf{x}$. The algorithm returns a boolean $b$. The simulator creates the share $\langle\!|b|\!\rangle_{\mathsf{P}}$, such that together with $\langle\!|b|\!\rangle_{\mathsf{S}}$ it would declassify to $b$.

The simulator expects $\mathcal{A}$ to command $\mathcal{M}_{\mathsf{S}}$ to submit $(\mathsf{declassify}, sid, \langle\!|b|\!\rangle_{\mathsf{S}})$ to $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$. The simulator simulates the interaction of $\mathcal{F}^{\mathrm{AP}}_{\mathsf{declassify}}$, making $\mathcal{A}$ learn $b$.

If $b$ happened to be 0, then the simulator tells $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to not proceed. If $b = 1$, then the simulator expects to receive $(\mathsf{ss}, sid, [\![\mathbf{z}']\!]_{\mathsf{S}})$ from the adversary. If $[\![\mathbf{z}']\!]_{\mathsf{S}}$ is equal to $[\![\mathbf{z}]\!]_{\mathsf{S}}$, then the simulator sends $(\mathsf{sig}, sid, \mathbf{z})$ back to the adversary

and allows $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to proceed. If $[\![\mathbf{z}']\!]_{\mathsf{S}}$ is different from $[\![\mathbf{z}]\!]_{\mathsf{S}}$, then the simulator sends nothing back to the adversary and tells the ideal functionality to not proceed.

Indeed, sending a different $[\![\mathbf{z}']\!]_{\mathsf{S}}$ corresponds to changing the signature $(\mathbf{z}, c)$ into $(\mathbf{z} + \mathbf{z}^{\delta}, c)$, where $\mathbf{z}^{\delta} = [\![\mathbf{z}']\!]_{\mathsf{S}} - [\![\mathbf{z}]\!]_{\mathsf{S}}$ can be computed by the simulator. The size of the individual elements of $\mathbf{z}^{\delta}$ is bounded by $2(\gamma_1 - \beta)$. For the changed signature to verify, the equality $(\mathbf{A} \cdot \mathbf{z})^H = (\mathbf{A} \cdot (\mathbf{z} + \mathbf{z}^{\delta}))^H$ must hold, or we have a collision of the hash function $H$. This equality implies that $\mathbf{A} \cdot \mathbf{z}^{\delta} \approx \mathbf{0}$, i.e. we have a short solution $\boldsymbol{x}$ for the task $(\mathbf{A} \mid \mathbf{I}) \cdot \boldsymbol{x} = 0$.

**Unsuccessful signing.** On input $(\mathsf{signature}, sid, \mu, \perp)$ from $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$, the simulator *Sim* proceeds as follows. It generates a random $\langle\!\langle \mathbf{w} \rangle\!\rangle$, secret-shares $\langle\!\langle \mathbf{w}^H \rangle\!\rangle$ and computes $\langle\!\langle \mathbf{w}^L \rangle\!\rangle \leftarrow \langle\!\langle \mathbf{w} \rangle\!\rangle - \alpha \cdot \langle\!\langle \mathbf{w}^H \rangle\!\rangle$. The simulator also makes up random $\langle\!\langle \mathbf{z} \rangle\!\rangle_{\mathsf{S}}$ and $\langle\!\langle \mathbf{x} \rangle\!\rangle_{\mathsf{S}}$. With these values in hand, the simulation proceeds identically to the "Successful signing" case (Sec. 5.2), up to the point where the adversary $\mathcal{A}$ has sent the description of the algorithm $\mathfrak{A}$ to $\mathcal{F}_{\mathsf{rej\_check}}$ (received by the simulator). The declassification of $\mathbf{w}^H$ does not help $\mathcal{A}$ to figure out whether it is in a real or in a simulated execution [28]. The simulator now applies $\mathfrak{A}$ to such $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{P}}$ and $\langle\!|\mathrm{x}|\!\rangle_{\mathsf{P}}$ that (1) wouldn't pass the rejection check together with $\langle\!|\mathbf{z}|\!\rangle_{\mathsf{S}}$ and $\langle\!|\mathbf{x}|\!\rangle_{\mathsf{S}}$, and (2) spell the message "You're in a simulation".

At this point, the algorithm $\mathfrak{A}$ "knows" that it is in an execution with $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ and *Sim*, not with real system consisting of $\mathcal{M}_{\mathsf{P}}$, (corrupted) $\mathcal{M}_{\mathsf{S}}$, and all the smaller ideal components. But $\mathfrak{A}$ does not have the ability to communicate this finding back to $\mathcal{A}$. It may return the boolean $b = 0$, in which case the unsuccessful signing would "successfully finish". The same "successful finish" would also take place for $b = 0$ in an execution with the real system. Or, $\mathfrak{A}$ may return $b = 1$, leading to the Phone determining that the Server has cheated in both systems. A way for $\mathcal{A}$ to distinguish between the two systems is to get $b = 1$ from $\mathfrak{A}$ and then try to change $[\![\mathbf{z}]\!]_{\mathsf{S}}$ so that the signature would be valid. In this case the real system would return the signature to the environment, while the ideal system would return $\perp$. But if the value of $b$ may be wrong, then $\mathcal{A}$ does not know yet whether it is in the "successful signing" or "unsuccessful signing" scenario. Hence $\mathcal{A}$ would be able to change the signature also when $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ has created an actual signature, the impossibility of which we have discussed above.

This discussion has already explained how the simulation for "unsuccessful signing" concludes: $\mathfrak{A}$ creates a bit $b$, this gets secret-shared as $\langle\!|b|\!\rangle$ and declassified. If $b = 0$, then the simulator tells $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to proceed with outputting $\perp$. If $b = 1$ then the simulator waits for the message $(\mathsf{ss}, sid, [\![\mathbf{z}]\!]_{\mathsf{S}})$, ignores its content and tells $\mathcal{F}_{\mathsf{ML\text{-}DSA}}$ to stop.

## 6 Evaluation

We have implemented Trilithium in Rust, and measured its performance. All measurements were performed on Lenovo laptop with AMD Ryzen 5 PRO

| Category | P→ S | S → P | CRP → S |
|---|---|---|---|
| ML-DSA-44 | 460.26 KiB | 460.26 KiB | 1.42 MiB |
| ML-DSA-65 | 880.11 KiB | 880.11 KiB | 2.66 MiB |
| ML-DSA-87 | 1852.12 KiB | 1852.11 KiB | 4.44 MiB |

**Table 1.** The volume of transmitted data between the parties during ML-DSA key pair generation. The correlated randomness is transmitted to the server in a single packet stream before the protocol is run.

| Category | P→ S *or* S → P (single direction) | | | CR |
|---|---|---|---|---|
| | Commitment | RejCheck | Total | |
| ML-DSA-44 | 56.85 KiB | 36.50 KiB | 93.36 KiB | 55.05 MiB |
| ML-DSA-65 | 73.09 KiB | 49.35 KiB | 122.43 KiB | 71.23 MiB |
| ML-DSA-87 | 97.45 KiB | 66.81 KiB | 164.25 KiB | 95.95 MiB |

**Table 2.** Volume of transmitted data between client and server during a signing attempt. The presented values are the average of P→ S and S → P communication.

5650U. Traffic control (`tc`) was used to add latency between the phone and the server. Our set-up of the three parties during the performance measurements imitates the intended deployment of our protocol: The Server and CRP will be located close-by (but under independent control), with a high-bandwidth and low-latency connection between them. The Phone is located elsewhere, and the latency of the connection between the Phone and the Server will be significant. As usual, the parameters of the connection between the Phone and CRP are not significant: the CR share for the Phone is generated from a single random seed. The CRP knows that seed and generates Server's CR share so, that it matches Phone's share.

While the communication between CRP and Phone is negligible (and the same seed can be used over many signings), the size of randomness sent by CRP to the Server is very significant. That randomness is dominated by the MAC shares, and depends on the number of MACs we have given to each value. We have chosen the numbers of MACs so, that the probability of guessing the MAC keys for a single modulus is at most $2^{-128}$ per try. This means 6 MACs for values shared over $q$, and more for smaller moduli.

The number of communicated bytes in key generation and signing is reported in Tables 1 and 2. These numbers were collected in the networking layer of our implementation. We see that they are very much in the realm of practicality. Table 3 gives the bandwidth requirements for sustained computation of signatures.

Our key generation protocol has 3 rounds, while the signing protocol has 14 rounds. We have benchmarked their execution, and report the running times in Tables 4–7. We have varied only the latency of the network, not the bandwidth, because the number of communicated bytes is small. Each reported number is the average of 100 runs. Again, the timings are practical.

As we saw, for higher latencies, we have considered running several signing sessions in parallel, in order to reduce the effect of rejections on latency. Based

| Category | P→ S | S → P | CRP → S |
|---|---|---|---|
| ML-DSA-44 | 404 KiB/s | 387 KiB/s | 234 MiB/s |
| ML-DSA-65 | 638 KiB/s | 607 KiB/s | 362 MiB/s |
| ML-DSA-87 | 647 KiB/s | 616 KiB/s | 370 MiB/s |

**Table 3.** The bandwidth required to produce one ML-DSA signature per second. The value is computed as the bandwidth required for a single signing attempt multiplied by the expected number of repetitions.

| Category | No latency + CR | 30 ms + CR | 100 ms + CR |
|---|---|---|---|
| ML-DSA-44 | 90 + 10 | 408 + 40 | 1342 + 110 |
| ML-DSA-65 | 114 + 36 | 650 + 66 | 1658 + 135 |
| ML-DSA-87 | 99 + 18 | 549 + 49 | 1677 + 119 |

**Table 4.** Mean time (in milliseconds) for key generation. Mean time for creating CR is given separately.

| Category | CR | Commitment | RejCheck | Total w/o CR |
|---|---|---|---|---|
| ML-DSA-44 | 288 | 141 | 104 | 245 |
| ML-DSA-65 | 430 | 155 | 155 | 310 |
| ML-DSA-87 | 587 | 181 | 183 | 364 |

**Table 5.** Mean time in milliseconds required for a signing iteration with no latency.

| Category | CR | Commitment | RejCheck | Total w/o CR |
|---|---|---|---|---|
| ML-DSA-44 | 479 | 523 | 418 | 941 |
| ML-DSA-65 | 976 | 544 | 431 | 975 |
| ML-DSA-87 | 974 | 531 | 475 | 1006 |

**Table 6.** Mean time in milliseconds required for 3 parallel signing iterations with 30 ms latency.

| Category | CR | Commitment | RejCheck | Total w/o CR |
|---|---|---|---|---|
| ML-DSA-44 | 551 | 1436 | 1010 | 2446 |
| ML-DSA-65 | 1042 | 1500 | 1069 | 2569 |
| ML-DSA-87 | 1370 | 1562 | 1098 | 2660 |

**Table 7.** Mean time in milliseconds required for 3 parallel signing iterations with 100 ms latency.

| Category | 1 | 2 | 3 | 4 | 6 | 8 |
|---|---|---|---|---|---|---|
| ML-DSA-44 | 2271 | 1984 | 1585 | 1631 | 1677 | 2109 |
| ML-DSA-65 | 3965 | 2742 | 2837 | 3035 | 3806 | 4099 |
| ML-DSA-87 | 3752 | 3249 | 3497 | 3716 | 4251 | 5077 |

**Table 8.** Mean time in milliseconds of signature creation in dependence on the number of parallel iterations with no latency.

| Category | 1 | 2 | 3 | 4 | 6 | 8 |
|---|---|---|---|---|---|---|
| ML-DSA-44 | 4078 | 2821 | 2516 | 2580 | 2692 | 2910 |
| ML-DSA-65 | 6662 | 5143 | 3886 | 4065 | 5090 | 4797 |
| ML-DSA-87 | 6605 | 4262 | 4975 | 4763 | 4671 | 6188 |

**Table 9.** Mean time in milliseconds of signature creation in dependence on the number of parallel iterations with 30 ms latency.

| Category | 1 | 2 | 3 | 4 | 6 | 8 |
|---|---|---|---|---|---|---|
| ML-DSA-44 | 9763 | 6704 | 5753 | 4830 | 4950 | 5080 |
| ML-DSA-65 | 15667 | 9422 | 7558 | 6582 | 6837 | 6956 |
| ML-DSA-87 | 11741 | 8211 | 7248 | 8065 | 7409 | 7656 |

**Table 10.** Mean time in milliseconds of signature creation in dependence on the number of parallel iterations with 100 ms latency.

on experiments, the best trade-off was achieved when executing 3–4 sessions in parallel, during which more than 50% of signatures are produced in the first run. The averages from these experiments are reported in Tables 8–10.

For ML-DSA-44, executions with 3 parallel sessions and latency of 30 ms managed to create a signature within 1.5 seconds in more than 50% of cases. A major part of the time is spent on the computation and transmission of CR from the CRP to the server. The correlated randomness can be precomputed, reducing the signing time to around 0.75 seconds.

## 7 Concluding remarks

In this paper, we presented Trilithium, a new two-party protocol (with CRP assistance) which allows Phone and Server collaboratively and efficiently produce ML-DSA signatures. It should not be too hard to generalize it to a larger number of parties and to different access structures, as long as these are supported by the underlying MPC protocol set. Indeed, the key generation and the high bits computation readily generalize; they can be re-stated to run on top of an *Arithmetic Black Box (ABB)* [25], as long as this ABB supports computations modulo $q$ and modulo 2, and conversions between the two moduli. Our rejection checks use a comparison protocol specifically for two parties; this protocol would have to be replaced in other settings.

In a certification-heavy ecosystem, the certifiability of implementations and deployments of protocols has to be considered. In case of Trilithium, one also has

to consider the protection of the CRP, and the size of data sent from the CRP to the Server. E.g. Smart-ID recorded a billion transactions in Baltic states in 2023 [15]; if all these had used Trilithium for ML-DSA-44 signatures, the average CRP-to-Server communication rate would have been $\approx$ 7.5 GB/s. While the CRP-Server-pairs could be replicated, the necessary bandwidth is still significant. A Hardware Security Module (HSM) in the form factor of a PCI Express (that easily supports such communication rates) expansion card may be a suitable platform for the CRP. The hosting of the Server party in a similar platform may satisfy both the performance and certification requirements, once these have settled for quantum-safe cryptographic service providers.

## 8 Acknowledgments

This work was funded by the Estonian Research Council under the grant number PRG1780.

## References

1. Alkadri, N.A., Döttling, N., Pu, S.: Practical Lattice-Based Distributed Signatures for a Small Number of Signers. In: Pöpper, C., Batina, L. (eds.) Applied Cryptography and Network Security. pp. 376–402. Springer Nature Switzerland, Cham (2024)
2. Attrapadung, N., Morita, H., Ohara, K., Schuldt, J.C.N., Tozawa, K.: Memory and Round-Efficient MPC Primitives in the Pre-Processing Model from Unit Vectorization. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. p. 858–872. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3517407, https://doi.org/10.1145/3488932.3517407 (2025-01-02)
3. Battagliola, M., Borin, G., Meneghetti, A., Persichetti, E.: Cutting the grass: Threshold group action signature schemes. In: Oswald, E. (ed.) Topics in Cryptology – CT-RSA 2024. pp. 460–489. Springer Nature Switzerland, Cham (2024)
4. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. pp. 169–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ Signature Framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2129–2146. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363229, https://doi.org/10.1145/3319535.3363229
6. Bogdanov, D., Laud, P., Laur, S., Pullonen, P.: From input private to universally composable secure multi-party computation primitives. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014. pp. 184–198. IEEE Computer Society (2014). https://doi.org/10.1109/CSF.2014.21, https://doi.org/10.1109/CSF.2014.21

[15] https://www.skidsolutions.eu/news/smart-id-expands-to-belgium/

7. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Sec. **11**(6), 403–418 (2012). https://doi.org/10.1007/S10207-012-0177-2, https://doi.org/10.1007/s10207-012-0177-2
8. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367. IEEE, IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00032
9. Boschini, C., Takahashi, A., Tibouchi, M.: MuSig-L: Lattice-Based Multi-signature with Single-Round Online Phase. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022. pp. 276–305. Springer Nature Switzerland, Cham (2022)
10. Buldas, A., Kalu, A., Laud, P., Oruaas, M.: Server-supported RSA signatures for mobile devices. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017. pp. 315–333. Springer International Publishing, Cham (2017)
11. Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: Virtual smart cards: How to sign with a password and a server. In: Zikas, V., De Prisco, R. (eds.) Security and Cryptography for Networks. pp. 353–371. Springer International Publishing, Cham (2016)
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA. pp. 136–145. IEEE Computer Society (2001). https://doi.org/10.1109/SFCS.2001.959888
13. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: Uc non-interactive, proactive, threshold ecdsa with identifiable aborts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 1769–1787. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3423367, https://doi.org/10.1145/3372297.3423367
14. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: Reif, J.H. (ed.) Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada. pp. 494–503. ACM (2002). https://doi.org/10.1145/509907.509980, https://doi.org/10.1145/509907.509980
15. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Two-party ecdsa from hash proof systems and efficient instantiations. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 191–221. Springer International Publishing, Cham (2019)
16. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold ec-dsa. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) Public-Key Cryptography – PKC 2020. pp. 266–296. Springer International Publishing, Cham (2020)
17. Celi, S., Escudero, D., Niot, G.: Share the mayo: Thresholdizing mayo. In: Niederhagen, R., Saarinen, M.J.O. (eds.) Post-Quantum Cryptography. pp. 165–198. Springer Nature Switzerland, Cham (2025)
18. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago,

Illinois, USA. pp. 11–19. ACM (1988). https://doi.org/10.1145/62212.62214, https://doi.org/10.1145/62212.62214
19. Chen, M., Doerner, J., Kondi, Y., Lee, E., Rosefield, S., Shelat, A., Cohen, R.: Multiparty generation of an rsa modulus. Journal of Cryptology **35**(2), 12 (Mar 2022). https://doi.org/10.1007/s00145-021-09395-y, https://doi.org/10.1007/s00145-021-09395-y
20. Cozzo, D., Smart, N.P.: Sharing the LUOV: Threshold Post-quantum Signatures. In: Albrecht, M. (ed.) Cryptography and Coding – 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11929, pp. 128–153. Springer (2019). https://doi.org/10.1007/978-3-030-35199-1_7
21. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: Spd$\mathbb{F}_{2^k}$: Efficient MPC mod $2^k$ for dishonest majority. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10992, pp. 769–798. Springer (2018). https://doi.org/10.1007/978-3-319-96881-0_26, https://doi.org/10.1007/978-3-319-96881-0_26
22. D'Alconzo, G., Flamini, A., Meneghetti, A., Signorini, E.: A framework for group action-based multi-signatures and applications to LESS, MEDS, and ALTEQ. Cryptology ePrint Archive, Paper 2024/1691 (2024), https://eprint.iacr.org/2024/1691
23. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer (2006). https://doi.org/10.1007/11681878_15, https://doi.org/10.1007/11681878_15
24. Damgård, I., Mikkelsen, G.L., Skeltved, T.: On the security of distributed multiprime rsa. In: Lee, J., Kim, J. (eds.) Information Security and Cryptology - ICISC 2014. pp. 18–33. Springer International Publishing, Cham (2015)
25. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 247–264. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
26. Damgård, I., Orlandi, C., Takahashi, A., Tibouchi, M.: Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. Journal of Cryptology **35**(2), 14 (Apr 2022). https://doi.org/10.1007/s00145-022-09425-3
27. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. pp. 643–662. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
28. Devevey, J., Fallahpour, P., Passelègue, A., Stehlé, D.: A detailed analysis of fiat-shamir with aborts. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V. Lecture Notes in Computer Science, vol. 14085, pp. 327–357. Springer (2023). https://doi.org/10.1007/978-3-031-38554-4_11, https://doi.org/10.1007/978-3-031-38554-4_11

29. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ecdsa from ecdsa assumptions: The multiparty case. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1051–1066 (2019). https://doi.org/10.1109/SP.2019.00024
30. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Secure two-party threshold ecdsa from ecdsa assumptions. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 980–997. IEEE (2018)
31. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ecdsa in three rounds. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 3053–3071 (2024). https://doi.org/10.1109/SP54263.2024.00178
32. Doran, R.W.: Variants of an improved carry look-ahead adder. IEEE Trans. Computers **37**(9), 1110–1113 (1988). https://doi.org/10.1109/12.2261, https://doi.org/10.1109/12.2261
33. Drake, J., Khovratovich, D., Kudinov, M., Wagner, B.: Hash-based multi-signatures for post-quantum ethereum. IACR Communications in Cryptology **2**(1) (2025). https://doi.org/10.62056/aey7qjp10
34. Drijvers, M., Edalatnejad, K., Ford, B., Kiltz, E., Loss, J., Neven, G., Stepanovs, I.: On the security of two-round multi-signatures. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1084–1101 (2019). https://doi.org/10.1109/SP.2019.00050
35. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 238–268 (2018). https://doi.org/10.13154/tches.v2018.i1.238-268
36. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12171, pp. 823–852. Springer (2020). https://doi.org/10.1007/978-3-030-56880-1_29, https://doi.org/10.1007/978-3-030-56880-1_29
37. Feng, Q., Yang, K., Zhang, K., Wang, X., Yu, Y., Xie, X.: Stateless deterministic multi-party EdDSA signatures with low communication. Cryptology ePrint Archive, Paper 2024/358 (2024), https://eprint.iacr.org/2024/358
38. Fleischhacker, N., Herold, G., Simkin, M., Zhang, Z.: Chipmunk: Better Synchronized Multi-Signatures from Lattices. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 386–400. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623219
39. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU (2018)
40. Garillot, F., Kondi, Y., Mohassel, P., Nikolaenko, V.: Threshold schnorr with stateless deterministic signing from standard assumptions. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 127–156. Springer International Publishing, Cham (2021)
41. Gennaro, R., Goldfeder, S., Narayanan, A.: Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security. In: Manulis, M., Sadeghi, A.R., Schneider, S. (eds.) Applied Cryptography and Network Security. pp. 156–174. Springer International Publishing, Cham (2016)

42. Gur, K.D., Katz, J., Silde, T.: Two-Round Threshold Lattice-Based Signatures from Threshold Homomorphic Encryption. In: Saarinen, M.J., Smith-Tone, D. (eds.) Post-Quantum Cryptography. Springer Nature Switzerland, Cham (2024)
43. Hazay, C., Mikkelsen, G.L., Rabin, T., Toft, T.: Efficient rsa key generation and threshold paillier in the two-party setting. In: Dunkelman, O. (ed.) Topics in Cryptology – CT-RSA 2012. pp. 313–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
44. Katsumata, S., Reichle, M., Takemure, K.: Adaptively secure 5 round threshold signatures from mlwe/msis and dl with rewinding. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024. pp. 459–491. Springer Nature Switzerland, Cham (2024)
45. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 830–842. ACM (2016). https://doi.org/10.1145/2976749.2978357, https://doi.org/10.1145/2976749.2978357
46. Khaburzaniya, I., Chalkias, K., Lewi, K., Malvai, H.: Aggregating and thresholdizing hash-based signatures using starks. p. 393–407. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3524128, https://doi.org/10.1145/3488932.3524128
47. Komlo, C., Goldberg, I.: FROST: Flexible Round-Optimized Schnorr Threshold Signatures, pp. 34–65 (07 2021). https://doi.org/10.1007/978-3-030-81652-0_2
48. Komlo, C., Goldberg, I.: Arctic: Lightweight and stateless threshold schnorr signatures. Cryptology ePrint Archive, Paper 2024/466 (2024), https://eprint.iacr.org/2024/466
49. Koziel, B., Gordon, S.D., Gentry, C.: Fast two-party threshold ecdsa with proactive security. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. p. 1567–1580. CCS '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3658644.3670387, https://doi.org/10.1145/3658644.3670387
50. Kravtšenko, S.: Efficient Two-Party ML-DSA Protocol in Active Security Model. Master's thesis, University of Tartu (2025)
51. Krenn, S., Lorünser, T.: An Introduction to Secret Sharing. SpringerBriefs in Information Security and Cryptography, Springer Cham (2023)
52. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (Feb 2014). https://doi.org/10.1007/s10623-014-9938-4, http://dx.doi.org/10.1007/s10623-014-9938-4
53. Lindell, Y.: Fast secure two-party ecdsa signing. Journal of Cryptology **34**, 1–38 (2021)
54. Lindell, Y.: Simple three-round multiparty schnorr signing with full simulatability. IACR Communications in Cryptology **1**(1) (2024). https://doi.org/10.62056/a36c0l5vt
55. Lindell, Y., Nof, A.: Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody. In: In the 2018 ACM SIGSAC Conference. pp. 1837–1854 (10 2018). https://doi.org/10.1145/3243734.3243788

56. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. pp. 598–616. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
57. Makri, E., Rotaru, D., Vercauteren, F., Wagh, S.: Rabbit: Efficient comparison for secure multi-party computation. In: Borisov, N., Díaz, C. (eds.) Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12674, pp. 249–270. Springer (2021). https://doi.org/10.1007/978-3-662-64322-8_12, https://doi.org/10.1007/978-3-662-64322-8_12
58. National Institute of Standards and Technology: SHA-3 standard:: permutation-based hash and extendable-output functions. National Institute of Standards and Technology (U.S.), Gaithersburg, MD 20899-8900 (2015). https://doi.org/10.6028/nist.fips.202, http://dx.doi.org/10.6028/NIST.FIPS.202
59. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology (U.S.), Gaithersburg, MD 20899-8900 (2024). https://doi.org/10.6028/nist.fips.204, http://dx.doi.org/10.6028/NIST.FIPS.204
60. Nick, J., Ruffing, T., Seurin, Y.: Musig2: Simple two-round schnorr multi-signatures. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 189–221. Springer International Publishing, Cham (2021)
61. Nick, J., Ruffing, T., Seurin, Y., Wuille, P.: Musig-dn: Schnorr multi-signatures with verifiably deterministic nonces. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 1717–1731. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3417236, https://doi.org/10.1145/3372297.3417236
62. Nicolosi, A., Krohn, M., Dodis, Y., Eres, D.: Proactive two-party signatures for user authentication. In: NDSS Symposium 2003 (12 2002)
63. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 681–700. Springer (2012). https://doi.org/10.1007/978-3-642-32009-5_40, https://doi.org/10.1007/978-3-642-32009-5_40
64. Peralta, R., Brandão, L.T.: NIST First Call for Multi-Party Threshold Schemes. National Institute of Standards and Technology (U.S.), Gaithersburg, MD 20899-8900 (Jan 2023). https://doi.org/10.6028/nist.ir.8214c.ipd, http://dx.doi.org/10.6028/NIST.IR.8214C.ipd
65. Pettai, M., Laud, P.: Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015. pp. 75–89. IEEE Computer Society (2015). https://doi.org/10.1109/CSF.2015.13, https://doi.org/10.1109/CSF.2015.13
66. Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: 2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001. pp. 184–200. IEEE Computer Society (2001). https://doi.org/10.1109/SECPRI.2001.924298
67. Reisert, P., Rivinius, M., Krips, T., Hasler, S., Küsters, R.: Actively secure polynomial evaluation from shared polynomial encodings. In: Chung, K., Sasaki, Y. (eds.) Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the

Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part VI. Lecture Notes in Computer Science, vol. 15489, pp. 3–35. Springer (2024). https://doi.org/10.1007/978-981-96-0938-3_1, https://doi.org/10.1007/978-981-96-0938-3_1

68. Snetkov, N., Vakarjuk, J., Laud, P.: TOPCOAT: towards practical two-party Crystals-Dilithium. Discover Computing **27**(1), 18 (Jul 2024). https://doi.org/10.1007/s10791-024-09449-2
69. Toft, T.: Primitives and Applications for Multi-party Computation. Ph.D. thesis, University of Aarhus (2007)
70. Veugen, T.: Encrypted integer division and secure comparison. Int. J. Appl. Cryptogr. **3**(2), 166–180 (2014). https://doi.org/10.1504/IJACT.2014.062738, https://doi.org/10.1504/IJACT.2014.062738
71. Xue, H., Au, M.H., Xie, X., Yuen, T.H., Cui, H.: Efficient online-friendly two-party ecdsa signature. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 558–573. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484803
72. Yao, A.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982. pp. 160–164. IEEE Computer Society (1982). https://doi.org/10.1109/SFCS.1982.38, https://doi.org/10.1109/SFCS.1982.38
73. Yuan, B., Yang, S., Zhang, Y., Ding, N., Gu, D., Sun, S.: MD-ML: super fast privacy-preserving machine learning for malicious security with a dishonest majority. In: Balzarotti, D., Xu, W. (eds.) 33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024. USENIX Association (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/yuan
74. Zhou, L., Wang, Z., Cui, H., Song, Q., Yu, Y.: Bicoptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 534–551. IEEE (2023). https://doi.org/10.1109/SP46215.2023.10179449, https://doi.org/10.1109/SP46215.2023.10179449