Almost-Asynchronous MPC under Honest Majority, Revisited

Matthieu Rambaud* and Antoine Urban*

Telecom Paris, Institut Polytechnique de Paris, France

Version 4 - 11/08/2021 [1]

Abstract. Multiparty computation does not tolerate $n/3$ corruptions under a plain asynchronous communication network, whatever the computational assumptions. However, Beerliová-Hirt-Nielsen [BHN10, Podc'10] showed that, assuming access to a synchronous broadcast at the beginning of the protocol, enables to tolerate up to $t < n/2$ corruptions. This model is denoted as "Almost asynchronous" MPC. Yet, their work [BHN10] has limitations: (i) *Setup assumptions:* their protocol is based on an encryption scheme, with homomorphic additivity, which requires that a trusted entity gives to players secret shares of a global decryption key ahead of the protocol. It was left as an open question in [BHN10] whether one can remove this assumption, denoted as "trusted setup". (ii) *Common Randomness generation:* the generation of threshold additively homomorphic encrypted randomness uses the broadcast, therefore is allowed only at the beginning of the protocol (iii) *Proactive security* is not guaranteed, enabling it comes with new challenges in this model. (iv) *Triple generation latency:* The protocol to preprocess the material necessary for multiplication has latency t, which is thus linear in the number of players. We remove all the previous limitations. The new protocols for (ii), (iii) and (iv) involve complexity tradeoffs, they are not mandatory for (i) removing the trusted setup.

1 Introduction

Secure multiparty computation (MPC) allows a set of n players holding private inputs to securely compute any arithmetic circuit over a (small) fixed finite field $\mathbb{F}_p$ on these inputs, even if up to t players, denoted as "corrupted", are fully controlled by an adversary $\mathcal{A}$ which we assume *computationally bounded*. MPC protocols in the *synchronous* model are extensively studied. The underlying assumption there is that the delay of the messages in the network is bounded by a *known* constant. However, the safety of these protocols fails when this assumption is not satisfied. Thus, protocols [DGKN09; HNP05; CHP13; BBCK14] were developed for the *asynchronous* communication model. This setting comes with limitations: Ben-Or, Kelmer, and Rabin [BKR94] proved that AMPC protocols are possible if and only if $t < n/3$, while we can tolerate $t < n/2$ in a synchronous environment. Moreover, Canetti [Can96] showed that it is impossible to enforce *input provision*, i.e. the inputs of all the (honest) parties are considered for the computation, which obviously, can represent an important setback for practical applications.

*Both authors contributed equally to the paper

[3] **Change log** w.r.t. Version 3 - 3 September 2021: Removed NIZK in the TAE and the non-proactive protocol of Thm 1, explicit succinct NIZK.

[4] **Change log** w.r.t. Version 4 - 8 November 2021: Univariate PVSS, UC Proofs, noninteractive proactive refresh, AAMPC from TFHE.

In [BHN10], Beerliová-Trubíniová, Hirt and Nielsen observed that, in a fully asynchronous network, assuming one initial synchronous broadcast round is sufficient to enforce input provision and tolerate $t < n/2$ corruptions. More precisely, they show that the minimum assumption is that players start with a consistent view of encrypted inputs. This model of [BHN08], which they denote "almost asynchronous" is relevant in, e.g., the use case where players would publish encryptions of their inputs on a public ledger, to demonstrate their interest in taking part to a MPC computation. Then, after a timeout (corresponding to an upper-bound on the publication delay of the ledger), the actual computation is done asynchronously on the published encrypted inputs. In their protocol, the circuit is evaluated using the *King/Slaves* paradigm [HNP05], in n parallel instances. Every player simultaneously acts as a king to evaluate its own computation instance with the help of the other players, and as a slave for other $n-1$ instances computing the same circuit. Their protocol assumes a mechanism by which players start with a *consistent view* on correct threshold encryptions of the inputs, in which all the honest inputs are taken into account. Players precompute threshold ciphertexts of multiplication triples, then perform additively homomorphic operations on these ciphertexts. This computation structure guarantees that every (recipient) player ultimately learns at least $t+1$ identical plaintext outputs of the circuit (with respect to the instances of honest kings), then terminates within a constant number of interactions. The problem is that, to implement the threshold additive encryption required in their protocol, they need that a trusted entity assigns secret keys to players ahead of the execution. But, under asynchrony, it is impossible to implement such a trusted entity with an asynchronous distributed protocol (see [AJM+21] for a state of the art) under honest majority. The reason is that Byzantine agreement is impossible beyond $t < n/3$.

It was left as an open problem how to remove this "trusted setup": in [BHN10, §4.3 *"our protocol requires quite strong setup assumptions, and it is not clear whether they are necessary."*]. The main contribution of the present paper is to remove it.

In detail, recall that a protocol is called *transparent* (or "ad hoc" [DHMR08; RSY21]) if it does not require a trusted setup phase, i.e., all public parameters are random coins. Our protocol has transparent setup, since we assume only a bulletin board ([BCG21]). As formalized in §2.3.1 and denoted $\mathcal{F}_{\text{Board}}$, this functionality enables each player $i \in [n]$ to write an arbitrary string of bounded size ahead of the execution, such that $\mathcal{F}_{\text{Board}}$ will make it visible to all players that i wrote this string. But $\mathcal{F}_{\text{Board}}$ does not perform any check on the strings (and allow players to re-write their strings): this is why $\mathcal{F}_{\text{Board}}$ is also denoted as "*bare / untrusted PKI*" in [GJPR21].

Theorem 1. *Consider $n = 2t+1$ players in an asynchronous communication network, of which t are maliciously corrupt by a polynomial adversary, in the untrusted PKI model, with access to one synchronous broadcast at the beginning of the execution. Then there exists a protocol that UC implements secure evaluation of any arithmetic circuit over any (not too large) finite field $\mathbb{F}_p$, with input provision. Both the latency, and communication complexity measured in the number of threshold ciphertexts, are the same as in [BHN10]. However our threshold encryption in this model has ciphertexts of size $O(n)$ (instead of $O(1)$ in the trusted setup of [BHN10]).*

In what follows we highlight the technical hurdles with respect to previous works, then overview the protocol of Theorem 1. Then in §1.2 we sketch how we solve the other limitations (*ii*) (*iii*) and (*iv*) mentionned in the abstract. Then in §1.3 we depict the ways in which our modular constructions can be combined with each other. Finally in §1.4 we outline a minor contribution, which is how recent works on threshold FHE can be adapted to fit into the almost-asynchronous model.

1.0.1 Roadmap of the Proof of Theorem 1 We first stress in §1.1.1, §1.1.2 that previously known transparent threshold encryption schemes support only a finite number of homomorphic additions, due to correct decryption being not guaranteed when plaintexts grow too large. Then, in §1.1.3, we evidence the difficulty to find an interactive mechanisms to bring down the size of the plaintexts. We then sketch in §1.1.1 the main novel ingredient that we introduce to solve Theorem 1. Namely: a threshold encryption scheme (TAE) with transparent setup, equipped with an interactive two-move mechanism (Resize) to enable an unlimited number of additively homomorphic operations. In §2.1 we detail the model, in §2.2 we recall the baseline protocol of [BHN10], in 2.3.1 we present our new transparent setup and in §2.4 we recall basic cryptographic primitives. In §3 we specify TAE, then in §3.4 we deduce the proof of Theorem 1 by recasting the baseline protocol with these new ingredients.

1.1 Main contribution: Threshold-Additive Encryption (TAE) with Transparent Setup

1.1.1 Previous Works: PVSS as Threshold Encryption with Transparent Setup Let us first recall what is a verifiable threshold encryption scheme. It is a public key cryptosystem between n fixed players, that comes with: a public algorithm, denoted **PubDec**.Contrib, that enables any of these players, on input a ciphertext and his secret key, to ouput a "decryption share" along with a ZK proof of correctness; and a public algorithm denoted **PubDec**.Combine that, on input any $t+1$ decryption shares, reconstructs the plaintext. This is often achieved assuming a trusted dealer, that publishes a global encryption key and and privately gives shares of the decryption key to players as their secret keys ([CDN01; CLO+13]). On the other hand, how to achieve threshold encryption *with the transparent setup* $\mathcal{F}_{\text{Board}}$ follows from an old idea. Namely: generate a secret sharing of the plaintext with threshold $(t+1)$, for instance with Shamir's scheme. Then, output the encryption of the shares under the public keys of the players (the i-th under the public key of the i-th player), along with a NIZK proof of correctness. This is suggested for the first time by Goldreich et al. [GMW91, §3.3], where it is presented as a scheme to verifiably share a secret in one single round of broadcast. Notice that, as detailed in §2.3.2, it is only recently that UC NIZK could be implemented under honest majority only using $\mathcal{F}_{\text{Board}}$. Remarkably, this has been independently re-discovered by three other research threads: first by [Sta96], in which it is formalized as *Publicly Verifiable Secret Sharing* scheme (PVSS), followed by [FO98; Sch99; YY01] [CS03, §1.1]; then rediscovered by Fouque and Stern [FS01, §4] as the main tool for a one-round discrete-log key generation protocol; and finally rediscovered as *threshold broadcast encryption* by Daza et al [DHMR08], followed by [CFY17] [RSY21, Appendix E].

1.1.2 Previous Limitations in the Number of Linear Homomorphic Operations, due to Growth of Size of the Plaintext Since we follow the blueprint of the MPC protocol [BHN10], we need to support homomorphic linear combinations on the ciphertexts, i.e., additions and scalar multiplications . The straightforward idea to achieve this is to instantiate the previous PVSS, with linearly homomorphic encryption. Unfortunately, all of the previous works which applied this idea, ended up with supporting only a limited number of linear operations. So this is incompatible with secure MPC evaluation of circuits of unlimited size. These works are either based on the additive variant of el Gamal, which we denote as "in-the-exponent", or the variant of Paillier encryption which we denote as "shifted to the negatives", which we both formalize in §2.4.3 and §2.4.4. Let us illustrate the roadblock encountered by these previous works.

el Gamal-in-the exponent the PVSS of [Sch99, §5], which is applied to electronic voting, is instantiated with a variation of el Gamal encryption, in which the plaintext is now encoded in the exponent of the ordinary el Gamal plaintext, which makes it additively homomorphic. We denote this variation as "el-Gamal in the exponent", see §2.4.4 for details. But in this scheme, decryption is performed by computation of discrete logarithm, which is a computationally hard task (although denoted by "can be computed efficiently" in [Sch99, bottom of page 11]). Since the size of the plaintext grows at each linear homomorphic operation, decryption is thus computationally untractable above a certain plaintext size, and thus, after a certain number of additions. This is stressed by [RSY21]: "In Shamir-and-ElGamal we are limited to polynomial-size message spaces since final decryption uses brute-force search to find a discrete log.".

Paillier The additive PVSS of [RSY21, Appendix E.2], is instantiated with Paillier encryption. Since players have *different* public keys N_i, they have different plaintext spaces $\mathbb{Z}/N_i\mathbb{Z}$. Thus, additions of PVSS are guaranteed to decrypt correctly only if the plaintexts' sizes do not wraparound in any of the $\mathbb{Z}/N_i\mathbb{Z}$, i.e., stay below the $\min(N_i)$. This is why it is said in [RSY21] that this PVSS *"supports a limited number (currently set to n) of homomorphic additions"*.

Contrast with trusted setup. Notice that this issue *does not happen when assuming a trusted setup.* For instance in the Paillier additive threshold scheme considered in [CDN01; BHN10], then all plaintexts belong to a fixed $\mathbb{Z}/N\mathbb{Z}$ with *unique* N. This guarantees unlimited homomorphic linear operations modulo N in this single ring of plaintexts. This enables their MPC protocols to directly evaluate circuits over $\mathbb{Z}/N\mathbb{Z}$.

Semi Homomorphic Encryption (SHE) Bendlin et al.[BDOZ11, Section 2] coined the notion of SHE, to denote any public key encryption scheme that supports a limited number of linear operations. Namely, a SHE guarantees correct decryption modulo p as long as the size of plaintext is below some bound M. Concretely, for Paillier, this requires to translate the plaintext space from $[0, \ldots, N-1]$ to $[-M, M]$, with $M := (N-1)/2$, in order to also ensure correct decryption modulo p after multiplication by -1: see §2.4.3 for details. But if the size of the plaintext grows above $M := (N-1)/2$, then the decryption modulo p is in general *not* equal to the plaintext modulo p. Although not considered in [BDOZ11], we observe in §2.4.4 that el Gamal in the exponent is also a SHE.

1.1.3 Our contribution: Maintaining Small Plaintexts Sizes under Asynchrony

Recall that a PVSS ciphertext is, itself, a vector of n ciphertexts of shares. To maintain the plaintexts of the PVSS shares of small sizes, and thus overcome the previously mentioned issues, one could think of the following mechanisms.

First attempt. At regular intervals, each honest player i would decrypt its share (the i-th) of the PVSS, reduce it modulo p to reduce the size of the plaintext (plaintexts being meaningful $\pmod p$), then re-encrypt it with its public key, and send it to the other players. This however fails in our *asynchronous* model. Indeed, a honest player i (even up to t of them) could be isolated from the network for an arbitrarily long time, while many noninteractive homomorphic additions are performed on the PVSS ciphertexts by the other players. Thus, the plaintext sizes of the i-th shares of the PVSS have grown very large. Thus when i is online again, he is unable to perform the correct decryption modulo p of his PVSS share, due to the limitation of SHE on the size of the plaintext for correctness of decryption modulo p.

Second attempt. In order to maintain the share s_i of each player i small, one can notice that the share of i, and thus its reduction $\bmod p$, of small size, is a linear combination $\bmod p$ of the shares of any other $t+1$ players (with the public Lagrange coefficients for polynomial interpolation). But, having naively each player j send the summand of the linear combination to i, would reveal the share of j to i. Thus, the linear combination needs to be done in an MPC manner, which involves *resharing*: we keep this idea for later.

Third attempt. One could think of the interactive mechanism, proposed by Choudhury-Loftus-Orsini-Patra-Smart, under the name "refresh" in Figure 3 of [CLO+13]. It consists of, on input a ciphertext X to be refreshed: collectively precompute a ciphertext Ma, denoted "mask", such that the plaintext ma is unpredictable uniform in the whole plaintext space; homomorphically compute the sum $Ma \boxplus X$; threshold decryption into $x+ma$; re-encryption of it then homomorphic subtraction of the mask Ma. However, applied in their context this mechanism leaves unchanged the plaintext: it reduces only the noise. In addition, their mechanism is *inapplicable* in the context of SHE because it requires to sum homomorphically the plaintext with a random mask, whose plaintext is sampled uniformly in *all* the plaintext space. Thus, the sum with the plaintext would go beyond the correct decryption bound M, thus decryption $\bmod p$ would be incorrect. Notice that in [CLO+13] this is correct, since their trusted setup allows an encryption scheme enjoying unlimited homomorphic additions $\bmod p$. Finally, a last issue of [CLO+13] is that they require a consensus on the masks used. In our model this would require that we generate the masks using the broadcast, which is costly.

Solution: interactive masking $\bmod, p$. We overcome both problems of the third attempt with the crucial observation the mask needs only be sampled uniformly in the *small subset* $\mathbb{F}_p := [0, \ldots, p-1]$ of the plaintext space. Indeed, since the SHE decryption is *modulo* p, this is sufficient for the decryption of the masked plaintext $(x + ma) \bmod p$ to vary uniformly in $[0, \ldots, p-1]$, and thus preserve privacy. Moreover, since $ma \in \mathbb{F}_p$, the final homomorphic subtraction of Ma keeps the plaintext in a small interval: $[-(p-1), \ldots, p-1]$, as desired. To address the last issue, on the consensus on the masks required in [CLO+13], we emphasize a major relaxation of their model. Namely, our masks are specific to each king/slaves instance. Moreover we even tolerate that a corrupt king sends *different* encrypted masks to two honest players. Hence we *do not need consensus* of the slaves on the masks returned by a possibly dishonest king: this is the same relaxation as the one observed in [BHN10] in the context of multiplicative triples. This allows us to adapt the triple generation mechanism of [BHN10] to build a slow king-dependent encrypted randomness generation protocol (detailed in §B.3). We denote Resize the overall mechanism, since it has for effect to reduce smaller than p the sizes of the plaintexts. All in all, such PVSS instantiated with SHE and equipped with this Resize is an instantiation of what we denote "threshold additive encryption" (TAE) with transparent setup, in §3.1.

Alternative Solution: bivariate PVSS, from any *PKE.* In §3.1.1 we remove the need for SHE and construct a TAE from any public key encryption scheme (PKE). The idea is given by the Second attempt above: a solution not to expose the individual shares to j, is that players compute the linear combination $\bmod p$ of their shares in an MPC manner before opening them to j. Precisely, a MPC linear combination of shares requires each player to *reshare* its share: we go one step beyond this idea, and construct directly TAE as a PVSS based on a bivariate sharing. Then, we have a generalization of the mechanism for reduction $\bmod p$, which enables to realize homomorphic

additions, and more generally linear combinations, in *one* interaction with the king, i.e., in two steps. This two-steps structure is the same as for any other threshold mechanism, e.g., for signatures ([ACR21]), decryption ([CDN01]) or common randomness ([CKS05]).

Epilogue: another tradeoff, with BGV. Let us make here the simple but apparently new observation that, instantiating PVSS with the encryption scheme of [BGV14], enables unlimited additions, without the need of any interactive Resize mechanism at all. Of, course multiplications still require interactions, because they involve threshold decryptions (of the inputs masked by the two first elements of a triple). The reason of the observation is that the scheme [BGV14] has a single plaintext space: $\mathbb{F}_p$ (embedded in a large lattice $\mathbb{F}_p^N$) independent of private keys, and that ciphertexts enjoy an unlimited number of noninteractive additions. However, BGV ciphertexts are very large, since typical lattice dimensions N are at least 2^{12}, in order to guarantee hardness of LWE.

1.2 Advanced contributions

1.2.1 Fast King dependent encrypted randomness generator We stated in §1.1.3 that, in order to periodically reduce the size of plaintexts, we need preprocessed ciphertexts of random values in $\mathbb{F}_p$, denoted as "masks", Using a straightforward simplification of the triple generation method of [BHN10] (detailed in §B.3), we can generate a encrypted random value from a chain of $t+1$ consecutive randomizations that is relative to a king. For each randomization, the latter collects $t+1$ signed contributions appended with a NIZK proof of correct encryption. However, generating an encrypted random value is much less constraining than generating a multiplication triple. Hence, we show in Section §4.1 that the interactions can be reduced to *one*-round trip (still without broadcast). For this, the main idea is to have the king simply collect and sum $t+1$ random encrypted values, appended with a non-interactive ZK (NIZK) proof of correct encryption. As at least one honest player contributed, the result is unpredictable to an adversary. Furthermore, as pointed above we do not need consensus of slaves on the masks returned by a possibly dishonest king. Thus, in contrast with the full-fledged new computation structure detailed later in §1.2.4, we *do not need* here to have one additional round-trip to collect $t+1$ signatures and guarantee unicity of the mask.

1.2.2 *Constant time* triples generation In order to multiply secrets, a mainstream approach, since Beaver [Bea91], consists in having players precompute random secret multiplication triples in an input-independent *offline phase*, that are later used in the so-called *online phase* to evaluate a circuit. This preprocessing is achieved asynchronously in [BHN10] at a cost of a number of consecutive interactions linear in the number of players. We bring this latency down from linear to a small constant, by leveraging the initial round of synchronous broadcast and an innovative method from Choudhury-Hirt-Patra [CHP13], that *extracts* fresh random triples from triples coming from different players. However, their method is inherently limited to $t < n/4$, due to usage of Byzantine agreement, i.e., consensus, on the set of input triples taken into account. We push this limit to $t < n/2$, thanks to two technical novelties, as detailed in §4.2. First, we require every player to append a NIKZ proof to the encrypted triple that it broadcasts, in order to prove its multiplicativity. Second, we make the following structural modification. Where, in [CHP13], the number of input triples taken into account in the extraction is *fixed* equal to $n-t$, by contrast we take into account *all* the $n-t+t' = t+t'+1$ correct triples broadcasted, where t' is the *variable* number of corrupted

players who broadcasted correct triples. This enables extraction of $(t+t')/2+1-t'$ unpredictable triples, and thus of *at least one*.

Thus, to evaluate a circuit with c_M multiplication gates, this triples generation requires each player to broadcast c_M encrypted triples, unlike the one of [BHN10] (§2.2), used in Thm 1, which is broadcast-free but has latency *linear* in n.

1.2.3 *On-the-fly* generation of threshold-additive encrypted randomness In [BHN10] the authors detailed only the computation of deterministic circuits. For randomized circuits, they suggested (p213) that players provide "additional random inputs". But recall that providing inputs requires the use of the broadcast channel. Thus their suggestion has broadcast complexity *linear in the number of random gates*. In section 4.3 we introduce an alternative that has broadcast complexity *independent* from the number of random gates, but exponential in n the number of players. Let us sketch the idea.

In a first attempt, one could think of building on the mainstream coin-tossing scheme introduced by Cachin et al. in [CKS05]. Recall that this scheme enables players to locally generate shares of a random coin. The problem is that these are *multiplicative* shares, namely, they live in the exponent of a group with hard discrete log. Thus, multiplicative reconstruction does not commute with computing additively homomorphic encryption.

Thus, we take instead advantage of the scheme introduced by Cramer et al. [CDI05], denoted as pseudo-random secret sharing (PRSS). PRSS enables each player to produce directly the Shamir share of a random value. The *linearity* of the reconstruction of Shamir, and the additive *homomorphic* property of TAE, make it possible to encrypt the Shamir shares obtained *locally* at each player, then apply Shamir's linear reconstruction *homomorphically* on these encrypted shares, to deduce an encryption of the reconstruction of the coin. Finally, we augment this scheme with ZK proofs to add the robustness which was missing in [CDI05].

Theorem 2. *(Informal) Assuming the same setting as in Theorem 1, one can do almost-asynchronous MPC in latency in $O(c_M)$ round-trips of messages* [2]*, in particular,* independently of the number of players*. The broadcast size is* independent of the number of random gates*, and is linear in the number of inputs c_I and c_M. The non-broadcast communication size is unchanged from Theorem 1. Instantiated in our* new computation structure *(§1.2.4 and §4.4), termination is guaranteed at the pace of the* fastest honest king *(while in [BHN10] it is conditional on the* slowest *honest king among $t+1$).*

1.2.4 Proactive Security Ostrovsky and Yung [OY91] introduced the notion of proactive security, in which the life span of a protocol is divided into separate time periods denoted "epochs". It is assumed that the adversary is *mobile*, in that it can change its corruptions, as long as at most t players are corrupt per epoch. In our model this brings three new difficulties.

First, in the protocol of Theorem 1, which follows the baseline of [BHN10], a freshly decorrupted player, with all his memory erased, could not take over the role of a king. Thus, the adversary can freeze the circuit evaluation after the first two epochs. The reason being that, to drive the protocol, a king must be convinced by its slaves, in zero-knowledge, that they did correctly their task since the beginning of the protocol.

[2] Here c_M denotes the number of multiplication gates

Second, even if players manage to refresh their encryption key pairs and reencrypt the decryption shares of ciphertexts at regular intervals, then, the adversary $\mathcal{A}$ may still use the new keys of newly corrupted players, to decrypt their share of a given ciphertext c, for which $\mathcal{A}$ may have previously gained knowledge of t other shares. Thus, within 2 epochs, it may gain knowledge of $t+1$ shares, which is enough to reconstruct the secret plaintexts.

Thirdly despite, different protocols, e.g., [CKLS02; SLL10; MZW+19] based on resharing have been proposed, following the seminal work of [HJKY95] on proactive security, they are *not* directly applicable in our setting as they either require broadcast or Byzantine consensus. Finally, the solution of [ZSV05] does not require consensus but has communication complexity *exponential* in n.

Necessity of a new computation structure To answer the aforementioned challenges, we first note that the computation model of [BHN10] does not guarantee the correctness of intermediary ciphertexts. Thus, a freshly decorrupted players who would like to resume the computation as a king, would have to start again from the beginning. We detail in §4.4 a new computation structure for proactive security, that creates a *chain of correctness* that guarantees the correctness of intermediary ciphertext and makes possible to resume the computation from the last step. Precisely, we make it possible for a new king to take over the role of a corrupted king, by enforcing the invariant that each correctly formed ciphertext output by a gate of the circuit, comes with *publicly verifiable* NIZK that they were correctly computed from the inputs. Players check these NIZK and sign the output if verification passes. Thus, the signature of any $t+1$ players on the ciphertext output of a gate constitutes a publicly verifiable certificate of correctness, which enables a new king to safely use it.

Furthermore, contrary to the computation model of [BHN10], in which a dishonest king may send *different* intermediary ciphertext outputs to its slaves, our new computation model enables a weak form of consensus on the intermediary ciphertexts output, thanks to a validation mechanism with the endorsement of $t+1$ signatures. The word "weak" is because the consensus is relative to each King/Slaves instance, and because it does not terminate if the king is dishonest. This allows us in §4.5.2 to adapt the resharing of [CKLS02] to rerandomize the PVSS shares, using only this weak form of consensus.

For simplicity, we first describe in §4.5.1 a simplified model, with a global clock and a synchrony assumption at every epoch change as in [CKLS02]. We adapt the refresh mechanism of [CKLS02] in our context of PVSS, as detailed in §4.5.2, and of weak form of consensus. Notice that we could have followed instead the alternative approach of [MZW+19], where players *interactively* generate a ciphertext c_0 of 0 then add it to previous ciphertexts, which has the same effect to rerandomize decryption shares. However, as already pointed in [CKLS02], this costs an *interaction*, during which players hold simultaneously their old and new keys, which is unsafe. The interaction may furthermore take an infinite time is the king is dishonest (see our §4.5.3 for a fix). By contrast, (our adaptation of) the mechanism of [CKLS02] enables players to erase their old key just after they performed a sequence of *noninteractive steps*, which takes an infinitesimal time. Subsequently, we discuss in §4.5.3 the difficulties of removing this synchrony assumption then remove it at the cost of an additional interaction. Then in §4.5.3 we sketch, from the encrypted randomness generator introduced above, a decentralized mechanism to refresh the keys, in particular which needs not access to the bulletin board and which goes at the actual network pace.

In summary, we obtain the following theorem.

Theorem 3. *(Informal) Theorem 1 and 2 can be improved with proactive security with $O(n^3)$ bits of communication per secret to be refreshed and per epoch.*

1.3 Fitting the components together

(1) As sketeched in 3.1.1, TAE can be implemented from any public key cryptosystem with plaintext spaces containing $\mathbb{F}_p$. This enables to do an unlimited number of additions, at the costs of an interaction for each linear combination and of threshold ciphertexts of size n^2, because of bivariate PVSS. (2) In §3.3 we explain how the use of a SHE enables to implement a TAE enjoying a number of noninteractive additions equal to M/p, where M is the (large) upper bound on the plaintext size up to which correct decryption modulo p is guaranteed, followed by only one interaction due to the Resize mechanism sketched in §1.1.3, thus improving the latency complexity. Furthermore, as stressed in §1.1.3, this implementation of TAE can be realized with classical n-sized PVSS, even with SHE, thus reducing the communication complexity by n.

Note that to be able to compute circuit of unlimited depth, a key ingredient is the Resize mechanism. This requires the precomputation of random masks, driven by each king. We adapt the triples generation mechanism of [BHN10] (3) to produce random masks from $t+1$ consecutive interactions. Later in §4.4.1 (4) we show that masks can be produced in just one interaction with the king, which is a simple example of our new computation structure.

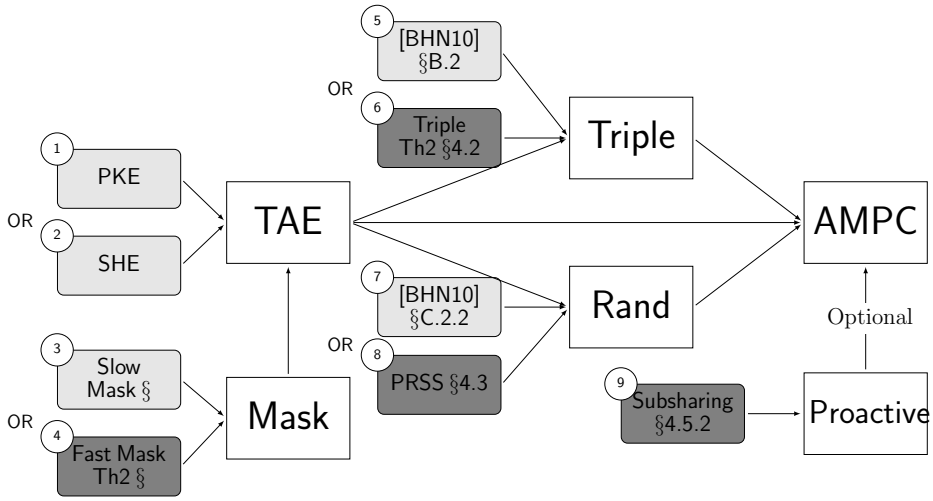

Fig. 1: Modularity of our contributions

Anticipating on the triple generation (presented in §4.2), we introduce a trade-off between the initial broadcast size and the communication complexity. The triple generation mechanism of [BHN10] (5) enables to produce multiplication triples on-the-fly at a cost of $O(n^2)$ interactions without using any initial broadcast, while our method (6) presented in §4.2 generates triples in a constant number of interactions but requires the broadcast of $O(c_M)$ elements, where c_M is the number of multiplication gates.

As noted by [BHN10, §3] *"any probabilistic polynomial-time functions can easily be computed by evaluating a deterministic function on the actual inputs and some additional random inputs provided by the parties."*. Such a Random gate, that produces a random number, uniformly and independently of everything else afresh for each invocation of the circuit, is implemented in [BHN10] (7) by broadcasting $O(nc_r)$) elements, where c_r is the number of random gates. This can turn to be prohibitive for some applications (gradient descent, ...). Thus, we show that leveraging PRSS [CDI05] (8), this can be implemented by broadcasting $O(exp(n))$, which is independent of c_r.

Finally, we introduce a last trade-off for proactive security. We propose a noninteractive solution for refreshing ciphertexts in §4.5.2 (9), using our implementation of TAE, inspired by the re-sharing technique of [CKLS02], at the cost of a $O(n^2)$ communication complexity per player. Note that if solution (8) is used to implement Random gates, then the PRSS keys have to be, in all cases, refreshed interactively.

1.4 Adapting Recent Works to the Almost Asynchronous Model

Following the initial result of [BHN10] to achieve security for an honest majority in the almost-asynchronous model, technical advances have been made. In §4.4.1 we analyze how recent works can be adapted to our almost asynchronous setting without trusted setup, and at which cost.

Following [DLS05] Recently, [DMR+21] proved that the threshold encryption scheme of [DLS05] allowed MPC in two rounds, of which only the first requires broadcast. It is in the model of $\mathcal{F}_{\text{Board}}$, augmented with a public uniform random string. The use of $\mathcal{F}_{\text{Board}}$ is explained in [DLS05] ("two rounds with PKI"). However it comes with the limitation that the second round is specified to be synchronous, unlike in our model. In addition, the size of their ciphertexts is polynomial in the depth of the circuit, thus limiting the depth of supported circuit. As a minor contribution, in §4.4.1 we observe that the first limitation is only apparent. Furthermore we observe that the interactive boostrapping mechanism of [CLO+13] is applicable and allows to regularly reinitialize the polynomial growth of ciphertexts. Importantly, as observed for our Thm 1, if casting the previous MPC protocol in the king/slaves computation model, then, the masks required by [CLO+13] *need not* anymore be generated by using the initial broadcast, since no consensus on the boostrapped ciphertexts is required, even not within a (dishonest) king/slaves instance.

Following [Sha17] First, and of independent interest, we observe in §4.4.1 a very *simple and alternative proof* of the recent feasability result of [GJPR21], of MPC in two rounds in the $\mathcal{F}_{Board}$ model without any public random string. Our construction is likely to have a different complexity tradeoff, since they use garbled circuits. Concretely, we replace the first round of input-independent broadcast of [BJMS20], which inherits from [Sha17], by a writing on $\mathcal{F}_{Board}$ in the (transparent) setup phase: this possibility was already observed by [GPS19a]. Now, to make the resulting scheme suitable for almost asynchronous MPC, it remains to remove the synchrony assumption in the final reconstruction round, and, also to prevent the polynomial growth of ciphertexts of [BJMS20] in both the number of players and in the depth of the circuit. This can be done exactly as in the previous paragraph.

But, let us observe that both previous solutions, using threshold FHE, have still their ciphertext sizes much larger than our TAE. Indeed, in addition to the polynomial dependency in depth, their ciphertexts initially consist of vectors of dimension the number of players, whose entries are of size quadratic in the number of players. In any case, recall that a minimum order of magnitude is that these entries are elements of lattices of dimension 2^{12} (for hardness of LWE).

Other recent works incompatible with our setting On the one hand, [Coh16] removes the need for the costly king/slaves communication paradigm, but requires the same trusted setup as [BHN10] for its threshold encryption. Also, [BBCK14] assumes trusted hardware to reach honest majority. Finally, the following two related papers accepted at TCC'21 (Iacr eprints 2021/1230 and 1233) have security with abort, whereas we guarantee output delivery.

2 Model and Definitions

2.1 Overall Goal

We consider $n = 2t + 1$ players $\mathcal{P} = \{P_1, \ldots, P_n\}$, which are a probabilistic polynomial-time (PPT) interactive Turing machines, of fixed and public identities. They are connected by pairwise authenticated channels. We consider a PPT entity denoted as the "adversary" who can take full control of up to t players, which are then denoted as "corrupt", before the protocol starts. For this reason we denote it as "static". Notice that a stronger adversary will be considered in §4.5. It can read the content of any message sent on the network. Being PPT, the adversary has however negligible advantage in the IND-CPA games that are satisfied by the encryption schemes considered.

2.1.1 Goal: Secure Computation of an Arithmetic Circuit over $\mathbb{F}_p$, with Input Provision Let us make precise the terminology used in Theorem 1. Let $p \geq n$ be any prime number, where n is the number of players defined above. We denote $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$ the finite field with p elements. For simplicity we state here the standalone security model, but will actually prove universal composability of our protocol. A MPC protocol takes as public parameter a fixed circuit $F : \mathbb{F}_p^n \to \mathbb{F}_p$ which is denoted as "arithmetic", in the sense that it is composed of addition gates, (bilinear or constant) multiplication gates (bilinear or constant, i.e., "scalar") and random values gates. For the sake of simplicity, we assume that all players are recipients of the final output. The *robustness with input provision* guarantee is that, for any set of inputs $x_i \in \mathbb{F}_p^n$, if each player starts with input x_i, then all players receive the same output y, and y is a (random) evaluation of $F(x'_1, \ldots, x'_n)$ such that $x'_i = x_i$ for all indices i of uncorrupted players. The *privacy* guarantee is that the adversary learns no more than y (and even nothing if no recipient is corrupted).

2.1.2 The Almost Asynchronous Model, after [BHN10] As observed in [BHN10], achieving a consistent view on ciphertexts of the inputs can be implemented assuming access to a synchronous broadcast channel at the starting time of the protocol (with delivery delay Δ): this is what they denote in [BHN08] the "almost asynchronous model". Any other related mechanism, e.g., a public ledger would also suit. Concretely, P_i broadcasts an encryption of its input, along with a call to a mechanism, which we denote $\mathcal{F}_{ZK}^{1:M}$ below, that guarantees that all players receive a proof of plaintext knowledge (PoPK). The most straightforward implementation of $\mathcal{F}_{ZK}^{1:M}$ consists in having P_i broadcast a NIZK PoPK.

But apart from the messages broadcast at $t = 0$, the network is otherwise fully asynchronous. Namely, messages sent by uncorrupted players are guaranteed to be eventually delivered, but the schedule is determined by the adversary.

2.2 Reminder of [BHN10, PODC'10]

Let us review in more details the MPC protocol of [BHN10] outlined in the introduction. It securely computes any arithmetic circuit over $\mathbb{Z}/N\mathbb{Z}$, with N a publicly known (large) integer, in the almost asynchronous model, and tolerates $t < n/2$ corruptions. Moreover it enforces *input provision*. Let us denote $\mathcal{E}$ an encryption scheme with plaintext space $\mathbb{Z}/N\mathbb{Z}$, that furthermore comes with a public *noninteractive* algorithm, denoted $\boxplus$, that computes the homomorphic addition of any two ciphertexts. Moreover, decryption of threshold ciphertexts is done by an interactive mechanism. Let F be the arithmetic circuit to be computed, which we assume deterministic here for simplicity. How to evaluate random gates will be discussed and improved in §4.3. The whole circuit is evaluated n in parallel, once for every player, denoted as *king*, and with all players (including the king), acting as its *slaves*.

(0) **Trusted Setup**: Taking as input the number of players $n = 2t + 1$, a trusted dealer publishes a public encryption key pk for $\mathcal{E}$, and sends privately a secret key sk_i to each player P_i.

(1) **Inputs broadcast**: Each player P_i broadcasts its encrypted input $\mathcal{E}_{\mathsf{pk}}(x_i)$ with a proof of plaintext knowledge. From now on, the communication pattern is asynchronous: each player waits for at most $t + 1$ correct messages from any $t + 1$ distinct players before sending new messages.

(2) **Triples generation**: is a subprotocol that enables players to jointly generate, with respect to a king, a multiplicative triple of encrypted values unknown to the adversary $\mathcal{A}$. The detail is that the king starts from a default known encrypted triple and sends a randomization request to every n slaves and waits for a valid answer. The king iterates this process a total of $t + 1$ times. Such a chain of $t + 1$ consecutive randomizations guarantees that the plaintext values of the factors of the encrypted triple, are indistinguishable to the adversary from uniform random ones. Details of the protocol in our model are presented in figure 5.

(3) **Circuit evaluation**: Each king P_j evaluates the circuit of F in a gate-by-gate manner, with the help of all players (including the king) acting as slaves. At each interactive step, the slaves prove to the king that their calculation is correct, which mainly consists in proving correct computation of their decryptions share. In particular, thanks to the multiplicative triples, the multiplication gates are brought down to threshold decryptions and homomorphic additions. This is an adaptation of the technique of [Bea91] over threshold ciphertexts, which we recast in Figure 6 in our model.

(4) **Termination**: Each encrypted circuit output, $\mathcal{E}_{\mathsf{pk}}(F(x_1, x_2, ..., x_n))$, is jointly decrypted to the king, which thus learns the plaintext result z. It sends z to all slaves. Players receiving z sign it and send the signature to the king. Upon receiving signature shares from $t + 1$ players, the king sends these signatures to all players. This guarantees unicity of one $(t+1)$-signed output per king. Once $t + 1$ kings have finished with the *same signed output*, then necessarily this must be the correct one, and all players adopt it.

2.3 The new transparent setup for Theorem 1

In 2.3.1 we formalize our setup, which is the untrusted PKI denoted as $\mathcal{F}_{\text{Board}}$. Then in 2.3.2 we specify the zero-knowledge functionalities that are implementable from $\mathcal{F}_{\text{Board}}$ (and from the first broadcast for $\mathcal{F}_{ZK}^{1:M}$) that we will use in our MPC protocol. In §2.3.3 we compare our setup with the trusted one of [BHN10]. In §2.3.2 we discuss other possibly instantiations of ZK-PoK under the additional assumption of a public *uniform* random string, which is still a setup known as *transparent*, by contrast with the trusted *structured random strings* required in SNARKS.

2.3.1 $\mathcal{F}_{\text{Board}}$ We formalize in figure §2 the public "bulletin board", denoted $\mathcal{F}_{\text{Board}}$, which is discussed in [BCG21], and also named as "bare/ untrusted PKI" in [GJPR21]. It is actually very close to the functionality denoted certification authority $\mathcal{F}_{CA}$ in Canetti [Can04]. Each player can write on this board an arbitrary string, whose maximal size is a public parameter. $\mathcal{F}_{\text{Board}}$ *does not* perform any check on the written value. Notice however that, in our protocol, players are instructed to broadcast a ZK proof of knowledge of their secret key, which thus has the same effect as the "KRK" functionality in [CDPW07]. As will be described in §2.3.2, implementing these ZK proofs is possible without further setup thanks to the honest majority setting. This will be required in our UC proof §B.6, when analyzing the view provided by the simulator, although our simulator *will not* extract secret keys from corrupt players. $\mathcal{F}_{\text{Board}}$ allows players to rewrite their string. $\mathcal{F}_{\text{Board}}$ enables every player to read the most recent string written by each player. Notice that in our protocol, honest players are instructed to write once (once per epoch, in §4.5).

$\mathcal{F}_{\text{Board}}$

Write On input (write, sid, v_i) from player P_i before the protocol starts, record the tuple (sid, P_i, v_i).

Read On input (read, sid, P_i) from player P_j, return the last tuple written by P_i: (sid, P_i, v_i), if any, or return $(sid, P_i, \perp)$ otherwise.

Fig. 2: Bulletin board functionality

To keep our proofs simple we disabled the possibility to (re)-write after the protocol starts (in §4.5: outside of the Refresh windows), although our Theorem 1 would remain true.

2.3.2 ZK implemented from $\mathcal{F}_{\text{Board}}$ During the transparent setup phase, apart from its public key, each player can also publish a random string, denoted as CRS. As observed in [GPS19b, §C.1], and credited to the [GO07], when honest players generate these CRS as specified in [GO07] under the name "multi-string CRS", then it is possible to implement a non-interactive zero knowledge proof system, noted NIZK for short, enjoying a property known as "simulation sound extractability". But as proven in [Gro06, §6.1], this property in turn implies that the NIZK implements $\mathcal{F}_{ZK}$ in the UC sense. [Notice that the word "sound" was dropped from the definition [GO07, p. 5] (and reappears page 9), this definition is nevertheless the same as the "simulation sound extractability" considered in [Gro06, §6.1].] Notice that this implementation of UC NIZK, from the previous mechanism denoted as "multi-string CRS", is specific to the honest majority setting. In the general case, a common uniform string is required to implement NIZK (see §2.3.4).

Next, although UC NIZK will be used in our advanced contributions §4.4, they are slightly overkill for the protocol §3.4.1 underlying Theorem 1. Thus, we describe the two weakenings that will be used in §3.4.1.

(Possibly interactive) zero knowledge. The zero-knowledge functionality $\mathcal{F}_{ZK}$ is presented in figure 3. It is *exactly the same* as defined in [CLOS02], noticeably, the adversary cannot trigger an abort. The same $\mathcal{F}_{ZK}$ is also used, unchanged, by [Coh16].

As stressed in [CLOS02], $\mathcal{F}_{ZK}$ actually specifies a proof of knowledge (PoK). Namely, if the verifer receives a message from the functionality proving that x belongs to the language, then, he is furthermore ensured that the prover knows a witness w, i.e., $R(x, w) = 1$.

$\mathcal{F}_{ZK}$

The functionality is parameterized with an NP relation R of an NP language L, running with a prover P, a verifier V and an adversary $\mathcal{A}$:

- Upon receiving $(\text{prove}, sid, x, w)$ from P, ignore if $R(x, w) = 0$. Otherwise send $(\text{verification}, sid, x)$ to $\mathcal{A}$ and V and halt.

Fig. 3: Zero-knowledge functionality

One-to-many zero knowledge. The following functionality, denoted $\mathcal{F}_{ZK}^{1:M}$ and presented in figure 4, was first introduced in [CLOS02, Figure 14], then also appears under the name $\mathcal{F}_{ZK}^{R}$ in the malicious compiler of [AJL+12, §E]. Noticeably, the latter compiler guarantees, in turn, the malicious security of [DLS05], revisited in the recent [DMR+21], that we fit in our model in §1.4.

$\mathcal{F}_{ZK}^{1:M}$

The functionality is parameterized with an NP relation R of an NP language L, running with a prover P, a set of verifiers $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$:

- Upon receiving $(\text{prove}, sid, x, w)$ from P, ignore if $R(x, w) = 0$. Otherwise send $(\text{verification}, sid, x)$ to $\mathcal{A}$ and all verifiers $P_1, \ldots, P_n$ and halt.

Fig. 4: One-to-Many Zero-knowledge functionality

In particular, it guarantees that either all honest players or none of them receive the proof within a *fixed* delay. This can trivially be implemented in the first round of broadcast, by having the prover broadcast a NIZK. Less trivially, it is observed in [BHN10, §5.1-5.2] that it can be implemented by what they denote as "almost-asynchronous NIZK". Namely, the prover broadcasts a string, which the players then interactively check as validly proving the claim. Unfortunately, their implementation of this original idea requires a trusted dealer of a shared secret between the verifiers, and thus is non-transparent.

Comparison with the weaker $\mathcal{F}_{ZK}^{1:M}$ of [Coh16]. Notice that a functionality, also denoted $\mathcal{F}_{ZK}^{1:M}$, appears in [Coh16], but is *strictly* weaker than the one defined above that we use. Indeed, this variant guarantees only that all players *ultimately receive* a proof, upon *querying* $\mathcal{F}_{ZK}^{1:M}$. But then, under asynchrony, a player may never know if the prover did not correctly called $\mathcal{F}_{ZK}^{1:M}$, or if the answer to his query is just taking a long time to arrive. This is why, in [Coh16], where this weaker $\mathcal{F}_{ZK}^{1:M}$ is used to prove plaintext input knowledge (as is the stronger $\mathcal{F}_{ZK}^{1:M}$ in our §3.4.1), then, contrary to us, *some honest inputs may never be taken into account.* Indeed, in [Coh16], players perform consensus on a set of $n - t$ inputs for which they received a proof of plaintext knowledge from the weak $\mathcal{F}_{ZK}^{1:M}$. This limitation can be seen more concretely in the implementation of $\mathcal{F}_{ZK}^{1:M}$ in [Coh16]. Namely, a prover is instructed to prove its claim to a quorum of players sufficiently large to contain at least a honest one. Then gather signatures from this quorum, which thus constitute a proof of validity of its claim. But a malicious prover may forward this proof, i.e. the signatures, only to *some* honest players, not all. Thus, unlike in our $\mathcal{F}_{ZK}^{1:M}$, there is no guarantee that either *all* honest players receive a proof or *none*. What is guaranteed by the implementation of [Coh16] (but not explicit), is that, if any (honest) player *requests* all players if they did receive a quorum of

signatures attesting validity of the proof, then, if at least one honest player received the quorum, then it will *ultimately* have the signatures forwarded to the requesting player.

2.3.3 Comparison with the setup of [BHN10] In [BHN10], players are *assigned* a (common) public key (and *correlated* private keys) by a trusted authority: this is what [BCG21, p9] denote as a *full blown trusted party*. If the adversary learns these secrets, then the security is ruined. By contrast, the $\mathcal{F}_{\text{Board}}$ that we use manipulates no secret information.

Let us also observe, although this is orthogonal to our concern, that all the instantiations of ZK proofs specified in [BHN10] also require a trusted setup, by contrast with the ones recalled in §2.3.2 and also in 2.3.4 (the latter requiring furthermore a public uniform random string). First, [BHN10, §5.1-5.2] assumes a secret sharing of a secret key by a trusted authority. Then, in [BHN10, §2.4], they use [Dam00], which requires a public string generated with a *secret trapdoor* that the adversary should not learn. This setup, also known as "structured common reference string" (also used in [GOS12] and SNARKS), is therefore not transparent. By contrast, the implementations of NIZK considered above in §2.3.2 do not require any public string.

By contrast, even the random string further required in the alternative implementations §2.3.4, is a mere public coin, thus is a transparent setup. Namely, it needs only be uniform, so *needs not* be generated with a secret trapdoor.

2.3.4 Alternative implementations of NIZK requiring furthermore a public uniform string Even though these are orthogonal to our work and in *no way* necessary for Theorem 1, let us discuss, for completeness, some instantiations of NIZK under the additional assumption of a public uniform string, which is still a transparent setup. How to implement a public distributed random beacon with transparent setup is well studied [CD20; Vin21]. We follow the example of the NIZK of [DDO+01], which enjoy many properties.

First, they implement non-interactively $\mathcal{F}_{ZK}$ in the strong sense of uniform composability (UC). This fact, which follows from the explicit simulation-soundness properties of their scheme, was observed independently by [CLOS02], [AJL+12, p497] and [Coh16, §4.2].

Second, they require only that players are provided with a *public uniform* random string, which *needs not* be generated with any trapdoor. This assumption, which would be necessary without honest majority, is also the one required by [AC20; ACR21]. Hence, without public random group elements, the Pedersen commitment scheme would simply not exist. This minimal assumption is also the one of the UC-NIZK implemented in [CSW19], and of the ZK-Starks [BBHR19], which is the proof system deployed in Ethereum. As nailed by the latter (footnote 8), randomness is actually necessary in any ZK proof system. See §B.1 for a concrete illustration of how to build a NIZK PoPK (and also a proof of correct re-encryption) for our TAE from the framework of [ACR21]. The proofs have logarithmic size in the statement proven. Their construction is conceptually as simple as opening $O(n)$ linear forms on a commitment in a DDH-hard group (noted additively). An application of the basic compression mechanism of [ACR21] then enables to compress the total size in $O(\log(n))$.

Third, the public uniform string can be safely *re-used* identically in multiple executions. Thus they achieve the stronger primitive which [CLOS02] formalize as the "multi-session" $\widehat{\mathcal{F}}_{ZK}$.

Notice that these implementations are provably UC in a *local setup*, where concretely a fresh random string is initially queried by players to the beacon. This enables the simulator to sample it with a trapdoor. As for [AC20; ACR21], in the noninteractive regime with Fiat-Shamir, simulatability can be proven if the random oracle is initialized by players in the setup, since then

the simulator can reprogram it. These assumptions are discussed in [Pas03; CDPW07]. Although discussing implementations of $\mathcal{F}_{ZK}$ is orthogonal to this work, for completeness we discuss in §B.7 implementations with a global setup.

2.4 Cryptographic primitives

2.4.1 Shamir secret sharing We denote $\mathbb{F}_p[X]_t$ the ring of polynomials with coefficients in $\mathbb{F}_p$, of degree bounded by t. Let us recall quickly the *secret sharing scheme of Shamir* over $\mathbb{F}_p$. We consider n fixed public nonzero distinct values in $\mathbb{F}_p$, denoted $[1, \ldots, n]$ for simplicity, denoted as the *evaluation points*. On input a secret $m \in \mathbb{F}_p$, sample at random a polynomial $f(X) \in \mathbb{F}_p[X]_t$, so of degree at most t, with nonconstant coefficients varying uniformly at random in $\mathbb{F}_p$, and such that $f(0) = m$, i.e., the constant coefficient is m. Then, output the n-sized vector $[f(1), \ldots, f(n)]$, denoted the "shares". It has the property that, for any fixed secret m, then any t shares vary uniformly (see Property 11 for details). While any $t+1$ shares *linearly* determine m as follows. For any subset $\mathcal{I} \subset \{1, \ldots, n\}$ of $t+1$ distinct indices, there exists $t+1$ elements $\lambda_i \in \mathbb{F}_p$, denoted the *Lagrange interpolation coefficients*, such that for every polynomial $f(X) \in \mathbb{F}_p[X]_t$ we have $f(0) = \sum_{i \in \mathcal{I}} \lambda_i f(i)$.

2.4.2 Semi-Homomorphic Encryption (SHE) Let us recall a simplified version of the definition of [BDOZ11, §2] of a public key Semi Homomorphic Encryption scheme (SHE). For simplicity we do not consider the limit on the size of noise in ciphertexts, since this limitation does not concern el Gamal in the exponent nor Paillier. A SHE is a tuple of algorithms (KeyGen, E, D) where: $\mathsf{KeyGen}(\kappa, p)$ is a randomized algorithm that takes as input a security parameter κ and a modulus p; It outputs a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$ and a set of parameters $P = (p, M, \mathbb{G})$. Here, M is an integer and $(\mathbb{G}, \oplus)$ is the abelian group where the ciphertexts belong (written in additive notation). For practical purposes one can think of M to be of size super-polynomial in κ, and p as being much smaller than M. We will assume that every other algorithm takes as input the parameters P, without specifying this explicitly. $E_{\mathsf{pk}}(x)$ is a PPT algorithm that takes as input an integer $x \in \mathbb{Z}$ and outputs a ciphertext $c \in \mathbb{G}$. Given $c_1 = E_{\mathsf{pk}}(x_1)$, $c_2 = E_{\mathsf{pk}}(x_2) \in G$, we have that $c_1 \oplus c_2$ is a ciphertext of $x_1 + x_2$, i.e., is an image of $x_1 + x_2$ under E_{pk}, which we thus also denote $E_{\mathsf{pk}}(x_1 + x_2)$. D_{sk} is a deterministic algorithm that takes as input a ciphertext $c \in \mathbb{G}$ and outputs $x_0 \in \mathbb{F}_p \cup \{\perp\}$. We say that a SHE with parameters $P = (p, M, \mathbb{G})$ is *correct* if, for all $x \in \mathbb{Z}$ such that $\left|x\right| \leq M$, then $D_{\mathsf{sk}}(E_{\mathsf{pk}}(x)) = x \bmod p$. We also require ind-CPA, with the usual definition.

2.4.3 Example of SHE: Paillier "shifted to the negatives" The Paillier encryption scheme, as e.g., recalled in [BDOZ11, §2.1], has plaintext space $\mathbb{Z}/N\mathbb{Z}$ with public key $\mathsf{pk} := N$ a large product of two secret primes, which themselves constitute the secret key, and ciphertext space $(\mathbb{Z}/N^2\mathbb{Z})^*$. Now, we modify the baseline Paillier encryption, in order to have decryption space $\mathbb{F}_p$ ($p \leq (N-1)/2$), with compatibility with taking the negative modulo p (and not N anymore), as follows. Let us denote $(\mathsf{KeyGen}, E_N, D_N)$ the baseline Paillier encryption with public key N, then we define: $E_N^{\mathsf{shift}} := E_N$ and, for every Paillier ciphertext c, we define $D_N^{\mathsf{shift}}(c)$ equal to: $D_N(c) \bmod p$ if $D_N(c) \in [0, (N-1)/2]$, or to $(D_N(c) - N) \bmod p$ if $D_N(c) \in [(N+1)/2, N]$. It follows that, for all x of size $\left|x\right| \leq M := (N-1)/2$, then $D_N^{\mathsf{shift}}\big(E_N^{\mathsf{shift}}(x)\big) = x \bmod p$, which is the SHE property.

Notice that, for our purpose of instantiating TAE with Paillier, we need that p be smaller than all the $(N_i - 1)/2$ for the public keys N_i generated with KeyGen. Thus, if a published public key

N_i is smaller than p, then players do as if P_i did not publish a key at all, as e.g., could happen if P_i is corrupt.

2.4.4 Example of SHE: ElGamal in-the-exponent The following scheme was used in [Sch99, §5] to instantiate a PVSS applicable to electronic voting. Notice that, although it supports a limited number of homomorphic additions, this scheme was not yet formalized, to our knowledge, as a "semi-homomorphic encryption" as defined in [BDOZ11]. Let $(\mathbb{G}, g)$ be a group of prime order q, along with a public fixed generator g. Precisely, we assume that the DDH problem is hard in $\mathbb{G}$, while g can be any element different from the unit. We denote $\mathbb{G}$ additively, as in [ACR21]. The plaintext space of the baseline ElGamal encryption is $\mathbb{G}$, which is isomorphic to $\mathbb{Z}/q\mathbb{Z}$. The ciphertext space is also $\mathbb{G}$. KeyGen is as follows. Sample $\mathsf{sk} \in \mathbb{Z}_q^*$ at random, and define $\mathsf{pk} := \mathsf{sk}.g$ as the public key (with a multiplicative notation, this would read g^{sk}). Now, to encrypt $\gamma \in G$ under public key pk, sample $r \in \mathbb{F}_q$ at random and output $(r.\mathsf{pk}, \gamma + r.g)$. Decryption is $\mathsf{Dec}\big(\mathsf{sk}, (ciph_1, ciph_2)\big) := ciph_2 - 1/sk.(ciph_1)$.

We modify this baseline scheme in order to obtain a decryption space equal to $\mathbb{F}_p$. We consider $\mathbb{F}_p$ as the subset $[0, \ldots, p-1] \subset \mathbb{F}_q$. For encryption, we map the input plaintext $x \to x.g$ (which reads g^x with a multiplicative notation), then apply the previously defined ElGamal. This is where our terminology "in-the-exponent" comes from. Now, decryption of a ciphertext $c \in \mathbb{G}$ consists in: applying the decryption of the baseline ElGamal to obtain some $x.g \in \mathbb{G}$, then try to compute the discrete logarithm x. (Notice that this step is denoted as "can be computed efficiently" in [Sch99], bottom of page 11.) Let M be a parameter such that discrete logarithm computation is efficiently computablen for exponents in $[-M, M]$, we thus need to assume $p \leq M$. If a discrete logarithm $x \in [-M, M]$ is found, then output $x \bmod p \in \mathbb{F}_p$. Else, output abort.

3 Proof of Theorem 1

3.1 Overview of transparent Threshold-Additive Encryption (TAE)

We sketch the TAE, following the program presented in §1.1.3. It is built from any SHE public key encryption scheme as defined in 2.4.2. Namely: a triple $(\mathsf{KeyGen}, E, \mathsf{Dec})$, with plaintext space $\mathbb{Z}$, and such that, on input a ciphertext of a plaintext m of size smaller than M, then Dec correctly returns m, up to modulo p (e.g., $M = (N-1)/2$ in the case of Paillier). The TAE has *transparent setup*, in that it takes as input any vector of (possibly empty) public keys $\mathbf{pk} = [\mathsf{pk}_1, \ldots, \mathsf{pk}_n]$. In the vector of public keys above, up to t can be empty ($\perp$), concretely, when corrupt players do not publish theirs.

To encrypt a plaintext $s \in [0, \ldots, p-1] \subset \mathbb{Z}$ with TAE, sample at random the nonzero coefficients of a symmetric polynomial B of degree $\leq t$, i.e., in $\mathbb{F}_p[X]_t$. Set the zero coefficient $B(0)$ as s. Finally, output the $\mathsf{TAE.ciphertext}$ c_s consisting of the vector of E-ciphertexts of evaluations of B: $E_{\mathsf{pk}_j}\big(B(j)\big)$. The homomorphic linear operations between $\mathsf{TAE.ciphertexts}$ are denoted with $\boxplus$ (addition) and $\boxdot$ (scalar multiplication), are defined by applying, entry-wise, the homomorphic addition and scalar multiplication of E. Correctness modulo p of $\boxplus$ (and likewise for $\boxdot$) below the size bound M, follows from the fact that the sum of two $\mathsf{TAE.ciphertexts}$ c_x and c_y, is the table of encrypted evaluations of the sum of the underlying polynomials, B_x and B_y, whose sum is B_{x+y}. Consider any ciphertext c, with plaintexts of shares below the size bound M. Any set of $t+1$ players can compute as follows the correct decryption mod p of c (the meaning of "correct" will be specified later, in terms of *well formed ciphertexts*.). Each player j computes the decryption of the

j-th coordinate of c, which we denote s_j: the *decryption share* of j. By definition of SHE we have $s_j \in [0, \ldots, p-1]$. We denote the result . Players send their decryption shares to all players (or, in our MPC protocol of §3.4.1: to the king, along with a correct decryption proof, then king-to-all). Upon reception of any $t+1$ of them, a player interpolates the unique polynomial $B \in \mathbb{F}_p[X]_t$ through the s_j received, and outputs the evaluation $B(0)$, which is equal to $s \bmod p$ as desired.

No Robustness We *will not* require for Robustness in our specification of TAE, namely, we do not specify that the algorithms listed in §3.2 come with a mechanism to verify correctness of their output. Notice that, by contrast, robustness is a requirement of *verifiable* threshold homomorphic encryption, as used in [CDN01].The reason for our choice is that robustness will be trivially enforced at the level of the MPC protocol, by having players ZK prove that they correctly computed their encryptions and their shares of the TAE operations. The reason for this modularity is that we need different levels of robustness. For the purpose of Theorem 1, slaves prove correctness in ZK only to their king K. But, as in the MPC protocol of [BHN10], kings do not prove to slaves the correctness of their computations, nor even of the inputs that they used. By contrast, in the proactive MPC protocol (§4), both slaves and kings will incorporate NIZK proofs of correctness in their shares and aggregation, until a quorum of $t+1$ slaves validate the proofs by putting their signature.

3.1.1 Alternative TAE from *any* PKE, with Interactive Linear Combinations In this paragraph we overview the alternative possibility to implement TAE from *any* public key encryption scheme, PKE in short, thanks to a novel *bivariate PVSS*, at the cost that linear combination are also computed interactively. To encrypt a plaintext $s \in [0, \ldots, p-1] \subset \mathbb{Z}$ with TAE, sample at random the nonzero coefficients of a symmetric bivariate polynomial B of bidegree $\leq (t,t)$, i.e., in $\mathbb{F}_p[X,Y]_{(t,t)}$. Set the zero coefficient $B(0,0)$ as s. Finally, output the TAE.ciphertext c_s consisting of the $n \times n$ array of E-ciphertexts of evaluations of B: $E_{\mathsf{pk}_j}\big(B(i,j)\big)$.

Interactive Additions for the Alternative TAE*.* The homomorphic linear operations between TAE.ciphertexts, and likewise for scalar multiplication or general linear combinations, proceeds by a two steps threshold mechanism. The mechanism maintains the invariant that, on each ciphertext, which is an $n \times n$ array, then at least $t+1$ lines are filled with values. On input two ciphertexts c_x and c_y to be added, each player j decrypts $t+1$ nonempty values x_i and y_i on the j-th column of each ciphertext. Then computes the $t+1$ sums of plaintexts $x_i + y_i$ modulo p, then interpolates the unique polynomial $B(X,j) \in \mathbb{F}_p[X]_t$ through these $t+1$ evaluations. He finally outputs its addition share, consisting in the row vector of encryptions of B at the n points $j' \in [n]$, under the public keys of players: $\big[E_{\mathsf{pk}_{j'}}(B(X,j'),\ j' \in [n]\big]$. On input any $t+1$ addition shares computed by distinct players, denote $\mathcal{I}$ the set of their $t+1$ indices, output the ciphertext sum consisting in the $n \times n$ array where the $t+1$ lines in $\mathcal{I}$ are equal to the shares received, and the other lines are empty (the entries are set to $\perp$).

Resizing for the Alternative TAE*.* We now sketch a threshold mechanism that inputs a ciphertext c, and output a ciphertext $c^{(out)}$ with same plaintext, but where in addition all the plaintexts of the entries in the $n \times n$ table constituting $c^{(out)}$ have been reduced mod, p, thus are back to small size. On input c, each player P_j decrypts $t+1$ entries on its column j into $d^{c}_{i \in [t+1],j}$, reduces them modulo p into $d^{c,(out)}_{i \in [t+1],j}$, interpolates the univariate polynomial $B_j(X)$ of degree $t+1$ through them, and deduces the remaining t evaluations $d^{c,(out)}_{i \in [t+1,\ldots,n],j}$. The key observation is that, by symmetry

of the bivariate polynomial B^c underlying c, we have that the $d^c_{i\in[n],j}$ are also the evaluations of B^c on the j-th row, i.e., at the points $(i, j \in [n])$. Finally, P_j outputs the j-th row, consiting in the re-encryption of the $d^{c,(out)}_{i\in[n],j}$ under the public keys $\mathsf{pk}_{i\in[n]}$. We denote this output a resize "share" of c. We then have the public algorithm takes as input any $t+1$ correctly computed Resize shares, and outputs, $n \times n$ ciphertext $c^{(out)}$, naively built as the $n \times n$ table in which the rows are set equal to the received shares if the case, and the remaining t rows are set empty $(\perp)$.

3.2 Specification of TAE

3.2.1 List of Algorithms Required Every algorithm for TAE takes as parameter the n public keys that are on the bulletin board. Since there is no condition on how the keys of malicious players were generated, nor any interaction nor trusted party needed to generate the keys, this setup is therefore *transparent*. The algorithms are straightforward to adapt for the cases where up to t keys are empty $(\perp)$. We consider a finite field $\mathbb{F}_p$ of prime order p. We fix E any SHE as defined in §2.4.2. We abuse notations and also denote as pk_i the public keys used for E . This abuse is because, in our implementation §3.3, E will be the baseline public key encryption scheme, thus the public keys will coincide. By "correct computation" of a decryption share from a secret key i, we imply in particular that the pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ was correctly generated with the KeyGen of E.

Definition 4. A $(t+1)$*-out-of* n *threshold encryption scheme with transparent setup* (TAE) over $\mathbb{F}_p$ is the data of a space $\boldsymbol{\mathcal{C}}$ denoted as the *global ciphertext space*, spaces $s\mathcal{K}$ and $p\mathcal{K}$ denoted as the "secret keys" and "public keys" spaces, of an integer M_{TAE} denoted the plaintext size bound for correct decryption $\bmod p$, and of the following algorithms. The $\{0,1\}^*$ denotes binary strings of which the lengths will be specified in our implementation. Λ denotes an arbitrary linear combination with arbitrary length L. Note the one can perform at once the threshold decryption of the linear combination of the plaintexts of ciphertexts.

Encrypt: $p\mathcal{K}^n \times \mathbb{F}_p \to \boldsymbol{\mathcal{C}}$

PubDec.Contrib**:** $s\mathcal{K} \times \boldsymbol{\mathcal{C}} \to \{0,1\}^* \cup \mathsf{abort}$, denoted as "decryption share".

PubDec.Combine**:** $(\{0,1\}^*)^{t+1} \to \mathbb{F}_p \cup \mathsf{abort}$.

$\boxplus$**:** $\boldsymbol{\mathcal{C}}^2 \longrightarrow \boldsymbol{\mathcal{C}}$

$\boxdot$ **:** $\boldsymbol{\mathcal{C}} \times \mathbb{F}_p \longrightarrow \boldsymbol{\mathcal{C}}$

Which furthermore satisfies *correctness of decryption* $\bmod p$*, privacy: IND-CPA and reconstructibility of decryption shares* as defined below.

In appendix §A.2 we further specify a private decryption algorithm, which is used for Theorem 1 only in the case where some outputs of the circuit are not to be learned by all players.

3.2.2 Correctness of decryptions mod p We firstly make a definition that characterizes a ciphertext obtained from repeated applications of $\boxplus$ and $\boxdot$. Let us consider a circuit C, with some arbitrary number L of input gates, all equal to **Encrypt**; with intermediary gates equal to $\cdot \boxplus \cdot$ and $\cdot \boxdot \lambda$ (the various factors $\lambda \in \mathbb{F}_p$ in the latter are public fixed values); and with one single output wire.

We then enrich the wires of circuit with labels, indicating an upper bound on the plaintext sizes through these wires. The fan-out wire of an **Encrypt** gate has size label $p-1$. The fan-out wire of

a $\cdot \boxplus \cdot$ gate has size label equal to the sum of the fan-in wires, while the one of a $\cdot \boxdot \lambda$ has size label equal to the product of the size label of the fan-in wire, times λ.

Let A be the size label of the output wire. Then, for any L inputs $(m_i)_{i\in[L]} \in \mathbb{F}_p$ (one for each input **Encrypt** gate), denote c the output of the circuit, we say that c is a *homomorphic linear combination of plaintext size* $\leq A$.

Definition 5. A $\mathsf{TAE.ciphertext}$ is a homomorphic linear combination of plaintext size $\leq M_{\mathsf{TAE}}$.

The *correctness* requirement is then that, for any $\mathsf{TAE.ciphertext}$ c, consider C be a circuit such as in the definition, i.e., with output size label $\leq M_{\mathsf{TAE}}$, and a collection of inputs $(m_i)_{i\in[L]} \in \mathbb{F}_p^L$, such that $c = C((m_i)_{i\in[L]})$ is the output, denote Λ_C the linear form $\mathbb{F}_p^L \to \mathbb{F}_p$ defined by the circuit C and denote $y := \Lambda_C((m_{i\in[L]}))$, i.e., the deterministic output of C if applied directly on the plaintexts, then, any public threshold decryption of c is equal to y.

In the sentence above, by any public threshold decryption, we mean: the output of $\mathbf{PubDec}.\mathsf{Combine}$ applied on any $t+1$ decryption shares, i.e., outputs of $\mathbf{PubDec}.\mathsf{Contrib}_{\mathsf{sk}_i}(c)$ computed with any $t+1$ secret keys sk_i with distinct indices.

We can thus say that y is *the* plaintext of the $\mathsf{TAE.ciphertext}$ c.

3.2.3 IND-CPA Is defined as usual. Consider a PPT adversary $\mathcal{A}$ playing with a challenge oracle $\mathcal{O}_{\mathsf{TAE}}$, which correctly generates $(\mathsf{sk}_i, \mathsf{pk}_i) := \mathsf{KeyGen}()\ \forall i \in [n]$ and gives all the pk_i to the adversary. Then, $\mathcal{A}$ can initially request "corruption" of at most t indices $\mathcal{I}_A \subset [n]$, for which $\mathcal{O}_{\mathsf{TAE}}$ reveals the $\mathsf{sk}_{i\in\mathcal{I}_A}$ to her. She can then make $\mathsf{TAE}.\mathbf{Encrypt}$ requests to $\mathcal{O}_{\mathsf{TAE}}$. $\mathcal{A}$ can submit two plaintexts m_0, m_1 to the $\mathcal{O}_{\mathsf{TAE}}$, who samples $b \in \{0,1\}$ and returns the encryption $c_i = \mathsf{TAE}.\mathbf{Encrypt}(m_i)$ to her. IND-CPA requires that $\mathcal{A}$ has negligible advantage in distinguishing b.

3.2.4 Reconstructibility of Public Decryption shares. This requirement, which we will use in the UC proof of Theorem 1 in §B.6, strenghtens the classical property known as "simulatability of decryption shares". It states existence of an efficient algorithm ShReco as follows. Consider the data of: *any* plaintext $m \in \mathbb{F}_p$, along with a subset of t indices $\mathcal{I}_{\mathcal{A}} \subset [n]$ and *any* t strings $(m_{i\in\mathcal{I}_{\mathcal{A}}})$ of the same format as decryption shares. Then, on input this data, we have that ShReco outputs $t+1$ strings $(m_{i\in[n]\setminus\mathcal{I}_{\mathcal{A}}})$, such that for any $\mathsf{TAE.ciphertext}$ of m such that the decryption shares produced by secret keys in $\mathcal{I}_{\mathcal{A}}$ are equal to the $(m_{i\in[n]\setminus\mathcal{I}_{\mathcal{A}}})$, then the decryption shares produced by the remaining keys are equal to the $(m_{i\in[n]\setminus\mathcal{I}_{\mathcal{A}}})$. We insist that ShReco does not take any secret key as input. Nor does the requirement depend on any ciphertext of m that would be taken as input of ShReco, as the case in usual definitions of "simulatability of decryption shares", e.g., [AJN+16, p. 17]. This requirement is what limited usual definitions: to simulatability of correctly computed decryption shares of a *given* input ciphertext. Thus these definitions did not enable a fake decryption (of a ciphertext, into an arbitrary plaintexts), which we need in our proof .

3.2.5 Inference of (Adversarial) Shares In the UC proof of Thm 1 in §B.6, contrary to [BHN10], the simulator does not know the secret keys of corrupt players. To cope with this, we require an algorithm ShReco that takes as input any $t+1$ decryption shares of some ciphertext c_m output by any $t+1$ secret keys of distinct indices, and outputs the t decryption shares of the remaining keys. We insist that ShReco does not take any secret key as input.

3.3 Implementation of TAE

We use the notations of §2.4. We consider a public key SHE scheme $(\mathsf{KeyGen}, E, \mathsf{Dec})$ over $\mathbb{F}_p$ as defined in §2.4.2, we denote M the size bound for correct decryption $\bmod p$. Given n public keys $\mathsf{pk}_1 \dots, \mathsf{pk}_n$, denote $(C_{i\in[n]})$ the corresponding ciphertext spaces. For brevity, we simplify the encryption notation $E(\mathsf{pk}_i, x)$ to $E_i(x)$. Then, we define the global ciphertext space $\boldsymbol{\mathcal{C}} := C_1 \times \cdots \times C_n$. We now introduce the following intrinsic definition. As stated later in Proposition 10, with respect to the following implementation of TAE, it will turn out that this definition is satisfied by any TAE.ciphertext (Definition 5), which will thus imply correct decryption $\bmod p$.

Definition 6. Let $A \leq M$ be an integer. A ciphertext $c = [c_1, \dots, c_n] \in \boldsymbol{\mathcal{C}}$ is a well formed ciphertext *with plaintext sizes smaller than* A if each c_i is a E_{pk_i}-ciphertext of some $s_i \in \mathbb{Z}$ with size $|s_i| \leq A$, which thus decrypts to $m_i := s_i \bmod p$ by the definition of SHE, and such that there exists a polynomial $B(X) \in \mathbb{F}_p[X]_t$, such that $s_i \bmod p = m_i = B(i)$. We then say that $m := B(0)$ is *the* plaintext of c.

Encrypt Let $(\mathsf{pk} \in p\mathcal{K}^n, m \in \mathbb{F}_p)$ be the inputs. Sample a random polynomial $B(X) \in \mathbb{F}_p[X]_t$, such that $B(0) = m$. Output $\big[E_i(B(i)),\ i \in [n]\big]$.

$\boxplus$ *and* $\boxdot$ are computed, coordinate by coordinate, by the $\boxplus$ and $\boxdot$ of the SHE.

Threshold decryption **PubDec**.Contrib: on input $(\mathsf{sk}_i,\ c \in \boldsymbol{\mathcal{C}})$, output $\mathsf{Dec}_{\mathsf{sk}_i}(c_i)$.
PubDec.Combine: on input (correct) decryption shares $m_{i\in\mathcal{I}} \in \mathbb{F}_p$ computed from $t+1$ secret keys with indices in any $(t+1)$-subset $\mathcal{I} \subset [n]$, interpolate the unique polynomial $B(X) \in \mathbb{F}_p[X]_t$ evaluating to them at the $[i \in \mathcal{I}]$, then output $B(0)$.

It follows easily from the description that the previous implemention of TAE satisfies correctness of threshold decryption modulo p, with respect to the same bound $M_{\mathsf{TAE}} := M$ as the correct decryption bound of the SHE. In addition to this fact, in §A.1 we also prove Reconstructibility of decryption shares and IND-CPA.

ShReco*:* on input a plaintext m and t elements $m_{i\in\mathcal{I}_\mathcal{A}} \in \mathbb{F}_p$ computed by secret keys with indices i in some t-subset $\mathcal{I}_\mathcal{A} \subset [n]$, interpolate the unique polynomial $B \in \mathbb{F}_p[X]_t$ evaluating to the $m_{i\in\mathcal{I}_\mathcal{A}}$ at the $[i \in \mathcal{I}_\mathcal{A}]$ and to m at 0. Then, output $B(i)$ for each $i \in [n]\backslash\mathcal{I}_\mathcal{A}$.

ShInfer*:* on input $t+1$ decryption shares $m_{i\in\mathcal{I}}$ computed by secret keys with indices i in some $(t+1)$-subset $\mathcal{I} \subset [n]$, interpolate the unique polynomial $B \in \mathbb{F}_p[X]_t$ evaluating to the $m_{i\in\mathcal{I}}$ at the $[i \in \mathcal{I}]$. Then, output $B(i)$ for each $i \in [n]\backslash\mathcal{I}$.

3.4 Proof of Theorem 1

3.4.1 Protocol We now present the overall protocol, that is further formalized in Appendix B. Note that we later introduce improvements for the triples generation (2) and the termination (4) in Sections 4.2 and 4.4 respectively, but they are in *no way* necessary for theorem 1. The structure is the *same* as the protocol of [BHN10], which was reminded in §2.2. We modify this baseline by using the above TAE with *transparent setup* instead of their additively homomorphic encryption with trusted setup.

We control the growth of size of plaintexts as follows. The wires of the circuit are labelled, as above Definition 5, by an upper-bound on the size of the plaintext. The previous rules of computation of the labels of fan-out wires are unchanged: **Encrypt** (has output label p), $\boxplus$ (addition of input labels), $\boxdot\lambda$ (multiplication with λ). In addition to the previous context of Definition 5, the MPC circuit has now also **PubDec** gates in different subprotocols (Multiplication, Termination and the new Resize that we are going to introduce). The output wires of **PubDec** gates are set to p. For instance, in the multiplication, the **PubDec** output wires are themselves input wires for the right input of $\cdot \boxdot \lambda$, i.e., define the public constants λ. For large circuits, labels might grow larger than M_{TAE}. To prevent this, and thus ensure correct decryption mod p, we also include a few Resize gates, as detailed in (3) below. Notice that this formalism parallels the one of so-called "well-formed-circuits" in [CLO+13], where in their context the labels are upper-bounds for the noise, instead of the plaintext size, and where special bootstrapping gates (denoted as refresh) are introduced to bring down the size of the noise, not of the plaintext size.

(0) **Transparent Setup**: Taking as input the number of players, each player generates locally a public/private key pair and sends the public key to $\mathcal{F}_{\text{Board}}$. Then it obtains all the submitted public keys. Likewise, players publish their individual CRS, following [GO07], in order to implement UC-NIZK (yielding $\mathcal{F}_{ZK}$, and $\mathcal{F}_{ZK}^{1:M}$ when combined with the broadcast).

(1) **Inputs distribution**: Players broadcast their encrypted inputs, and call $\mathcal{F}_{ZK}^{1:M}$ to provide all players with a proof of plaintext knowledge (PoPK) along with a proof of knowledge of their secret keys. See in §B.1 for an efficient noninteractive PoPK using [ACR21], which although assumes a common uniform string, which is not mandatory in our honest majority setting if using instead [GO07]. Upon completion of these steps, for every input wire, the players have agreement on: either a correctly formed ciphertext $X = \mathsf{TAE}.\mathbf{Encrypt}(x)$ of an input value, or on an empty string ($\perp$).

(2) **Triples generation**: The structure of this step is only slightly adapted from [BHN10], of which we recall the triple generation for convenience in Figure 5 of §B.2 (formalized in the $\mathcal{F}_{ZK}$-hybrid model).

(2) **Masks generation:** In order to implement Resize gates (detailed below), we need random masks. We detail in §B.3 an adaptation of the triple generation of [BHN10] to produce such random values.

(3) **Circuit evaluation**: The difference is that homomorphic linear combinations, which were computed on $\mathcal{E}-$ciphertexts, are now replaced by $\boxplus$ and $\boxdot$ on TAE.ciphertexts. This difference applies in particular to the Multiplication subprotocol. Note that the evaluation of a circuit require $O(c_M)$ consecutive interactions, where c_M denotes the number of multiplication gates.

(3) **Resize gates:** To resize a ciphertext, players use a fresh random mask that is subtracted from the plaintext using $\boxplus$. Then they compute the threshold decryption, in $\mathbb{F}_p$. Then the king re-encrypts it and subtracts the mask using $\boxdot(-1)$ then $\boxplus$. We detail this protocol in §B.5.

(4) **Termination**: This step is unchanged from [BHN10].

In §B.6 we prove that the protocol implements secure circuit evaluation, in the uniform composability (UC) sense. Since UC-NIZK are implementable from $\mathcal{F}_{Board}$ in the honest majority setting (by [GO07]), we make the proof in the $(\mathcal{F}_{Board}, \mathcal{F}_{ZK}^{1:M}, \mathcal{F}_{ZK})$-hybrid model, concretely we allow simulator to simulate all these functionalities. However, we *do not* allow the simulator to "rewind" the adversary, i.e., to extract witnesses from these two functionalities that it simulates. As in [BHN08, B], we *do* allow rewinding in *intermediary* games, whose purposes are to *analyze* the distribution of the view provided by the simulator, not to define the simulator. The main difference with [BHN10] (and also [CDN01; Coh16]) is that, there, $\mathcal{S}$im had the additional power to simulate the trusted setup, and therefore learn (even choose) the secret keys assigned to corrupt players.

He could therefore compute the decryption shares of corrupt players, then use this information to simulate *compatible* honest decryption shares of any plaintext of its choice. Our main innovation compared to [BHN10] is a way around that enables the simulator to still infer decryption shares of corrupt players despite its weaker power in our setting, thanks to the algorithm that we denoted ShInfer: *inference of adversarial shares*.

3.5 Complexity analysis

Let c_I, c_O and c_M the number of input, output, and multiplication gates. All ZK proofs with transparent setup considered in 2.3.2 have size at most linear in the circuit to be proven. Furthermore, those of [AC20; ACR21] have size logarithmic in the circuit to be proven, and can be made noninteractive by Fiat-Shamir. Therefore, we will omit the ZK proofs from the complexity analysis. We now analyze the overall protocol complexity. The communication complexity is measured as the number of TAE.ciphertext sent: in our implementation of TAE with transparent setup they are of size $O(n)$, whereas in the implementation of [BHN10] with trusted setup they are of size $O(1)$. We neglect the cost of Resize gates (and of the masks generation), since they happen once avery M/p noninteractive additions.

Input distribution (synchronous) In the first step, players broadcast their TAE.encrypted inputs, along with a NIZK PoPK. Thus, the size of the input to the broadcast channel is $O(c_I)$.

Triple generation from [BHN10] (asynchronous) The generation of a triple requires $O(n^2)$ randomizations (n in parallel, n times), even if only $O(n)$ randomizations are actually taken into account in the triple output. Each randomization is sent to all players, which thus involves n messages. A basic optimization presented in [BHN08, Appendix A.2] allows to reduce the amortized number of randomizations to n. Since each of the n kings need one triple per multiplication gate, the total communication complexity of generating triples is $O(c_M n^3)$.

Circuit evaluation (communication size) The gates $\boxplus$ and $\boxdot$ are performed locally, thus require no interaction. Each multiplication gate requires two **PubDec** (of the masked inputs). Players in **PubDec** communicate a total of $O(n^2)$ elements: 1 share sent to the king per player, times n players, times n kings.

Termination from [BHN10] The termination step requires each encrypted circuit output to be jointly decrypted to the king using **PubDec**.Combine, which then sends the plaintext result z to all slaves. Each player receiving z signs it and sends the signature to the king. Upon receiving signature shares from $t+1$ players, the king sends these signatures to all players. This guarantees unicity of one $(t+1)$-signed output per king. Once $t+1$ kings have finished with the *same signed output*, then necessarily this must be the correct one, and all players adopt it. The total communication size for decryption is $O(n^2)$ (as **PubDec**). Then the termination process in itself communicates $O(c_O n^3)$ elements.

4 Overview of advanced contributions

We now discuss some extensions to the Main Theorem 1. This section is intended to give a high-level overview of our more advanced results. Most details can be found in the appendices. Specifically, we

first detail in §4.1 a new constant-round protocol to generate a random value related to a king that we use for Resize. Then, we outline in §4.2 a new triple generation protocol used to prove Theorem 2. Then, we show in section 4.3 a new method for on-the-fly encrypted randomness generation. Finally, we detail in §4.4 an extension of the communication model of [BHN10] presented in §2.2 to achieve proactive security as detailed in section 4.5.

4.1 Fast King dependent encrypted randomness generator $\mathsf{TAE}.\boldsymbol{Rand_k}$ used in Theorem 2

Recall that, thanks to the observation that there needs not be consensus on the encrypted masks used in the Resize gates, we could generate them by adapting the method of [BHN10] for triples, as detailed in Appendix §B.3. but this took $t+1$ round trips between slaves and their kings. We now show that this can be reduced to one round trip, still without using broadcast. We introduce the following two-steps $\mathsf{TAE}.\mathbf{Rand}_k$ to generate a random value shared within a king/slaves instance. It can be seen as a toy example of our new computation structure (formalized in §4.5).

contrib$_{Rand_k}$**;** Each player P_i samples $m_i \leftarrow \mathbb{F}_p$ and send $M^{(i)} = \mathbf{Encrypt}(m_i, \mathsf{pk})$ to the king P_k along with its signature and a ZK proof of plaintext knowledge.

combine$_{Rand_k}$**:** Upon receiving messages from any subset $t+1$ slaves with correct proofs, the king outputs their $\boxplus$ sum, appended with the $t+1$ signed messages.

players receiving such an output M accept it if: it is appended with $t+1$ signed messages $M^{(i)}$ from different players with valid PoPK, and such that M is their $\boxplus$ sum.

Proposition 7. *If a honest player* accepts *such an output M, then its threshold decryption (**PubDec**), in $\mathbb{F}_p$, is unpredictable to the adversary $\mathcal{A}$.*

(Proof sketch). As soon as at least one player contributed to M (with an additive contribution $M^{(i)}$), we have that the threshold decryption of M, in $\mathbb{F}_p$, is unpredictable. □

4.2 Triple generation in constant round trips used in Theorem 2

We outline below our new alternative triples generation protocol, stated in Theorem 2 denoted as *PreProc*, that allows all the honest players to terminate the protocol and to output $\frac{t+t'}{2} + 1 - t'$ random multiplication triples unknown to the adversary. *PreProc* is detailed in appendix C.1.1 (fig. 7), and its proof in C.1.3. *PreProc* is independent of the threshold additive encryption scheme, e.g, which could either be the one considered in [CDN01; BHN10], which requires trusted setup, or ours in §3, which does not. Thus, we adopt generic notations: $\mathcal{E}$ denotes any threshold encryption scheme that enables (possibly with interactions) the addition of ciphertexts (noted $\boxplus$) and the scalar multiplication (noted $\boxdot$). *PreProc* is in three steps as follows:

1. **Triple distribution** In the initial broadcast round, each player P_i broadcasts one or several triples, encrypted with a threshold additive scheme, each of them appended with NIZK proofs of plaintext multiplication (PoPM). The relation to be proven is formalized as R_{PoPM} in §C.1.3, and can be directly and efficiently implemented in size logarithmic in the number of triples, e.g., with the technique recalled in [ACR21, §6].
2. **Triples verification** Each player then verifies the correctness of the multiplication triples and outputs the set $\mathcal{U}$ of the players who broadcasted correct multiplicative triples. Let us denote

$|\mathcal{U}| := t+1+t'$ their number[3]. Notice that, by contrast to [CHP13], this verification is local and deterministic, thus all honest players output the *same* $\mathcal{U}$ without needing Byzantine Agreement.

3. **Randomness extraction** Finally, each player executes the triple extraction protocol $TripExt$, presented in Appendix C.1.2, on the set of triples broadcasted by players in $\mathcal{U}$ to extract $\frac{t+t'}{2} + 1 - t'$ random multiplication triples unknown to $\mathcal{A}$. Our protocol is adapted from the protocol for the transformation of t-shared triples proposed in [CHP13], but unlike theirs, considers a *variable* number of encrypted triples, appended with a noninteractive PoPM. This enables the extraction of at least one triple without consensus. In short, the idea is to define two polynomials $\mathbb{x}(.)$, $\mathbb{y}(.)$ of degree $\frac{t+t'}{2}$, and to sacrifice $\frac{t+t'}{2}$ triples to interpolate a polynomial $\mathbb{z}(.)$ of degree $t+t'$, where $\mathbb{z} = \mathbb{x}(.)\mathbb{y}(.)$ holds. As $\mathcal{A}$ knows t' input triples, he learns t' distinct values of $\mathbb{x}(.)$, $\mathbb{y}(.)$ and $\mathbb{z}(.)$, implying $\frac{t+t'}{2} + 1 - t'$ degrees of freedom that can be used to extract random triples.

This protocol only involves one broadcast of $O(c_M)$ elements and one interactive decryptions during the extraction. We refer the reader to Appendix C for further details.

4.3 On-the-fly Threshold-Additive Encrypted Randomness used in Theorem 2

We propose a linear threshold construction to generate TAE encrypted random values without setup, such that players have a *consistent* view on the ciphertexts, using only a *constant* broadcast size. This leverages the construction introduced by [CDI05, §4] and denoted *pseudorandom secret sharing* (PRSS) and our TAE. The former enables players to generate, without interaction, an unlimited number of shared unpredictable random values, that come in the form of Shamir shares, that players generate locally. We introduce the following theorem.

Theorem 8. *In the same model than Theorem 1, the player can produce encrypted random values unknown to the adversary, in a fix (constant) number of consecutive interactions.*

The former enables players to generate, without interaction, an unlimited number of shared unpredictable random values, that come in the form of Shamir shares, that players generate locally.

We give details of Theorem 8 in appendix C.2 as well of security proofs. In brief, the main idea of this construction is to combine the linearity of the PRSS with the homomorphic properties of the TAE. This enables to generate key pairs on-the-fly which would be used in section 4.5 to enable proactive security.

4.4 New Computation Structure

We present the new computation structure to evaluate a circuit introduced in §1.2.4.

Stage, and speedup wrt [BHN10] We break down the actual computation of a circuit into a series of intermediary functions denoted as Stages. They represent the incompressible steps in our protocol and are entirely defined by a public *Stage Identification tag* (SID) as follows. The identity of the king is encoded as $SID.kingNb$. The function to be computed is denoted as $SID.function$. Finally, $SID.prev$ contains a list of SID's whose outputs are used as input of this stage. A stage takes as inputs outputs from previous stages and produces an output that we call a ***verified stage***

[3]Recall that t' denotes the number of correct triples broadcast by the adversary.

output (***Verif0ut*** in short), which consists of two elements: the result of the function $SID.function$ applied to the inputs from $SID.prev$ and a ***Quorum Verification Certificates*** (QVC in short) which consists in the concatenation of $t+1$ signatures on the result. Given a ***Verif0ut***, we use ***Verif0ut***.value and ***Verif0ut***.QVC to refer to the above-mentioned elements. Throughout the computation, we maintain the following invariant from the distribution to the termination:

$$Inv\_stage: \text{any output of a stage signed by at least } t+1 \text{ players is a correct verified stage output.} \tag{1}$$

This essentially forms a *chain of correctness* from distribution to termination.

Remark 1. Note that a player cannot terminate until it knows that all honest players will also terminate. In [BHN10], this requires every player to wait until they receive $t+1$ identical results to be sure that at least one honest king learns the correct result. In our protocol a signed value is correct (per Inv_stage). Upon receiving *one* correct output a player multicasts it and immediately terminates.

Overall structure of a Stage In summary, a stage takes as inputs a set of ***verified stage outputs*** $\{X_i\}_i$ and produces another ***VerifOut*** whose value is equal to $SID.function(\{X_i\}_i)$. The computation of this output follows a threshold mechanism in two steps.

contrib$_{sid}$**:** [4] a contribution function for stage SID applied by each slave P_j on: the stage input and its private input, denoted $\mathbf{s_j}$, typically its secret key.

combine$_{sid}$**:** a public function applied on any $t+1$ correct contributions, such that:

from *any* set $\mathcal{S}$ of $t+1$ slaves, we have

$$\textbf{\textit{Verif0ut}}.value = combine_{sid}(\{contrib_{sid}(\{X_i\}_i, s_j\}_{j\in\mathcal{S}}). \tag{2}$$

The execution of a Stage for a player is presented in figure 9 in Appendix D, along a more complete description of the data structures used and the pseudocodes.

A king drives a Stage in two exchanges of messages called *phases*. The first one is denoted as *contribution phase*. Each slave computes locally the function $contrib_{sid}$ on the inputs of the stage along with its secret input s_j. It sends the result, denoted "contribution" to the king, appended with a NIZK proof of correctness. Upon receiving $t+1$ correct "contributions" from $t+1$ distinct slaves, appended with the NIZK proofs, the king Combines these $t+1$ contributions into the stage output, and appends to it a concatenation of the proofs (denoted ***Combine Proof***).

In the second one, denoted *verification phase*, the king multicasts the above stage output appended with the concatenation of the proofs. Upon receiving it, each slave checks correctness of the NIZK then signs the stage output if correct. The king collects any $t+1$ signatures then concatenates them. Notice that this could be reduced to logarithmic size, thanks to the threshold signature without trusted setup of [ACR21, §5]. It appends them to the stage output: altogether, this constitutes what we denote the ***verified stage output***. Noticeably, these $t+1$ signatures by themselves guarantee that *at least one* honest player checked correctness of the stage output, and thus guarantee its correctness. Remarkably, this is why this data structure ***Verif0ut*** *needs not* to be further appended with the previous $t+1$ NIZK proofs to guarantee its correctness. Finally note that this verification mechanism, enable any new king to take over the computation, from the point where a former king became corrupt and stopped, which serves to enforce proactive security.

[4] To simplify notation, here *sid* denotes $SID.function$

Remark 2 (Optimization). At first glance, it seems that it takes 2 round trips for each operation. However, similarly to the "pipelining" technique of [YMR+19], one can have players speculatively execute the stages on some unsigned outputs of the previous stage while they are simultaneously performing the verification phase on these outputs. They abort if it turns out that these outputs cannot pass the verification. This halves the latency of a stage down to one round trip.

4.4.1 Fitting recent works in the almost asynchronous model Let us take the example of the MPC protocol [DLS05], the same observations and contributions apply to [BJMS20]. In their §5 they describe how to realize MPC in 2 rounds, from a "PKI setup" which we observe is actually equivalent to our $\mathcal{F}_{Board}$. Notice that, [DLS05, p. 9] also assume the setup of a public uniform random matrix denoted $\mathbf{B}$. This is not mandatory in [BJMS20], but their protocol requires 3 rounds. We now make here the simple but apparently new observation that their first round, of synchronous broadcast, can be *replaced* by a call to $\mathcal{F}_{Board}$ in the setup. Namely, and we credit this observation to [GPS19a], each player i in [BJMS20] can instead use $\mathcal{F}_{Board}$ to publish the input-independent material, denoted $param_i$, that it was intended to broadcast in the first round of [BJMS20]. As a consequence, we observe that this modification yields an *alternative proof of the result* of [GJPR21], which is that MPC is feasible in two rounds in the $\mathcal{F}_{Board}$ model, without any public random string.

Now, there remains an apparent separation between the previous MPC in two rounds, and the almost-asynchronous model. It is that the protocol of [DLS05] requires synchronous broadcast in the second round, which makes it incompatible with our model. To overcome this, let us recall that the general result of [DMR+21, §6 Thm 7] states that broadcast is not required in the 2^{st} and last round, as long as players are instructed to append PoPKs to their messages in the 1^{st} round. We now make the simple but apparently new observation that *even synchrony is not required* in the 2^{st} and last round, and thus that AMPC is possible. Indeed, it is stated at the end of the protocol of [DLS05, p11] that players are able to output in the last round from any set of $t+1$ correctly formed messages. Thus, this implies correctness of the modification of [DLS05, p11] in which we specify that the 2^{st} and last round is now *asynchronous*, i.e., that players output *as soon as* they receive any set of $t+1$ correctly formed messages. Indeed, whatever the scheduling of messages, the output of honest players is identical than in the baseline synchronous termination, since, in this baseline, each player takes only into account an arbitrary chosen subset of $t+1$ well-formed messages.

Then, the main limitation of the TFHE schemes [DLS05; BJMS20] presented in §1.4, is their ciphertext size polynomial in the depth of the circuit. Our second contribution here is to remove this limitation, using a mechanism similar as in [CLO+13]. Recall that they describe an interactive protocol, denoted "Refresh", as known as "interactive bootstrapping", that brings down the size of noise of threshold ciphertexts c. Concretely, players first collectively generate a random ciphertext, denoted as the "mask" c_r. Specifically, c_r is the homomorphic sum of random ciphertexts *synchronously broadcast* by players. Players then homomorphically add the mask to the ciphertext c_r to be refreshed, open $r+m$, compute a re-encryption of it then homomorphically subtract c_r. The problem is that both *synchrony and broadcast* are required, in order to guarantee consensus on the mask. Our observation is that, in our model, consensus on ciphertext masks need only hold *inside* every king/slaves instance. This *weak form* of consensus can be precisely implemented in our new computation structure by having the king choose which $t+1$ random contribution to sum to form c_r, then collect $t+1$ signatures on c_r to guarantee its *unicity*. We detail more this subroutine, denoted $\mathsf{TAE}.Rand_k$, in Appendix 4.1.

4.5 Proactive security

In the model denoted as "proactive", the execution is divided in timeframes, denoted as "epochs", such that the adversary can statically corrupt at most t players at the beginning of an epoch. Thus, to prevent the adversary to gain too much knowledge, players perform what we describe below as Refresh operations between every two epochs.

For simplicity we first define in §4.5.1 a model with epochs determined by a global clock, and with the synchrony assumption of [CKLS02] that all messages are delivered at the end of each epoch. How to implement a —more realistic— weak clock is well studied topic, e.g., see [NK20] for one with linear communication. In §4.5.2 we describe a mechanism to refresh ciphertexts with *fix latency*, and *no interaction* before the old keys are erased. Then in §4.5.3, we remove the synchrony, and adapt the refresh mechanism in consequence, at the cost of one more interaction. Finally in §4.5.3 we also sketch a decentralized mechanism to refresh the keys, in particular which needs not access to the bulletin board and which goes at the actual network pace. Note that we also provide further explanations about how our model stands compared to previous works [SLL10; BELO14; CKLS02] in E.1.

4.5.1 Simple Model and Overall Structure of a Non-interactive Refresh We assume a global clock that ticks epoch numbers: $e = 1, 2, \ldots$. Furthermore, the clock also regularly ticks another signal, that we denote as "beginning of Refresh". This signal is ticked a fixed public delay $\delta_{\mathsf{Refresh}}$ before every next epoch. The adversary can possibly decorrupt players at the beginning of Refresh. Corrupt players during Refresh leak twice more information to the adversary. Indeed, we will see that players during a Refresh know both secret informations relative to the current epoch e and to the next epoch $e+1$. Thus, if a player is corrupt during the refresh at the end of some epoch e, then we make the classical convention that the player also counts in the corruption budget relative to epoch $e+1$ (whose limit is also t).

Let us outline the different steps of a Refresh: Upon being notified a Refresh in epoch e, players generate a new key pair for epoch $e+1$ and publish the public key on the bulletin board. It is assumed that, within a known delay $\delta_{Board} \leq \delta_{\mathsf{Refresh}}$, all newly published keys are made accessible to all players by the bulletin board. We also make the synchrony assumption that, within a known delay $\delta_{sync} \leq \delta_{\mathsf{Refresh}}$, all messages sent in epoch e prior to the Refresh are delivered. From this point, players which did not publish a new key are considered as corrupt and their key is set to $\perp$. When $\max(\delta_{Board}, \delta_{sync})$ is elapsed, players *locally* perform a sequence of actions on the TAE.ciphertexts encrypted with the keys of epoch e. In detail, each slave, for each king/slave instance, gathers all TAE.ciphertexts that will be needed to finish the evaluation of the circuit, i.e., not the ones that were already processed through intermediary gates. Then it computes on each TAE.ciphertexts what is denoted as a ReShare.Contrib, as will be detailed in the next paragraph, then multicasts it. These actions being *noninteractive*, they thus take an infinitesimal time in our complexity model where latency is measured only in terms of communication delays.

Upon finishing ReShare.Contrib, players *erase* from their memory all their data excepted the keys of epoch $e+1$ (their secret key, and the n public keys). Therefore, when the clock ticks $e+1$, players have no more sensitive information relative to epoch e in memory.

Finally, all players acting as kings (including the ones which were freshly decorrupted), apply a public algorithm ReShare.Combine, that takes as input any $t+1$ correct ReShare.contrib of the *same* TAE.ciphertext (of epoch e), and which outputs a TAE.ciphertext encrypted with the keys of epoch $e+1$. Then they multicast these new TAE.ciphertext to their slaves, which enables them to take over the computation of the circuit.

Kings not only process the ReShare.Combine of their own slaves, but also of all slaves of all king/slaves instances (recall that the same player plays the role of n slaves in parallel). Therefore, this enables kings to take over the computation from the set of TAE.ciphertexts issued by the slaves of the king which was *the most advanced* in the computation of the circuit: this explains the last claim of Theorem 2. Notice that if we had not made the synchrony assumption, then, even in a honest king instance, some slaves may send $\mathsf{Refresh}_k$.contrib of older ciphertexts, i.e. outputs of previous gates, than more advanced other slaves. Thus, the king would never receive a set of $t+1$ contributions out of the same ciphertext, thus the computation would be stuck, because slaves would since have erased their old keys. in §4.5.3 we discuss further issues, and a solution for removing this assumption.

4.5.2 ReShare of TAE.ciphertexts, in our implementation §3.3 of TAE We recall that a well formed ciphertext c_s of some secret plaintext $s \in \mathbb{F}_p$, is a vector of encryptions of evaluations of a polynomial $B \in \mathbb{F}_p[X]_t$ such that $B(0) = s$.

Before describing our solution, let us recall the structure of the resharing mechanism of [CKLS02], that we use as inspiration. For convenience, we simplify it into semi-honest security and thus non-verifiable plain Shamir sharing. The initial state is that each player P_i owns a share s_i of a secret s, under univariate Shamir sharing with threshold t.

1. *Non-interactive resharing of each share:* Every player P_i randomly generates a polynomial $B_i(Y)$ of degree t evaluating to s_i at 0, and sends to each P_j the evaluation $s_i^{(j)} := B_i(j)$. Then P_i *immediately deletes* its share s_i.

Players perform multi-valued Byzantine consensus to select a subset $\mathcal{I}$ of $t+1$ indices of players of which the sharings have successfully terminated. Let $\lambda_{i\in\mathcal{I}}$ denote the Lagrange interpolation coefficients associated to $\mathcal{I}$.

2. *Generation of a new share:* Each player P_j computes its new share $s'_j := \sum_{i\in\mathcal{I}} \lambda_i s_i^{(j)}$.

This indeed defines a sharing of s, since the s'_j are evaluations at j of the polynomial $\sum_{i\in\mathcal{I}} \lambda_i B_i(Y)$, whose value at zero is equal to $\sum_{i\in\mathcal{I}} \lambda_i s_i = s$. We adapt this idea to our context of publicly verifiable secret sharing (PVSS), where shares come as public ciphertexts. But apart from this intuitive change, there is the nontrivial hurdle that Byzantine consensus *is impossible* under our honest majority setting. To overcome it, we observe that in our king/slaves computation model, the resharing of the PVSS needs only be consistent *within a King/slaves instance*. Then, using our new computation structure, makes it possible to enforce a king-dependent *consistent* multivalued consensus on a subset of $t+1$ contributors. Namely, the king chooses a subset $\mathcal{I}$ of $t+1$ contributions and requests signatures of players on this a subset. Upon collecting $t+1$ signatures, this guarantees unicity of this subset within a king/slave instance. The words "consistent" and "king-dependent" are here to stress that this weak form consensus, which is *relative to* a king/slave instance, is *not guaranteed to terminate* if the king is dishonest.

In appendix §E.2 we formalize the adaptation of the resharing mechanism of [CKLS02] in our context of PVSS, in the form of a threshold mechanism denoted $\mathsf{ReShare}_k$, where k stands for "king-dependent". Namely, step 1. above is adapted as $\mathsf{ReShare}_k$.contrib, then the linear combination of step 2. is now performed homomorphically on PVSS ciphertexts, which we formalize as $\mathsf{ReShare}_k$.Combine. Then in §E.3 we will state more precisely the following proposition and prove it.

Proposition 9. *The view of $\mathcal{A}$ during the* Refresh *operation is computationally independent of the plaintext s and of its view in previous epochs.*

4.5.3 Advanced model: Proactivity with responsive refresh window The refresh window is a weakness point in the protocol, since the corruption budget of the adversary counts twice. In this subsection we observe that we can improve security by making the duration of this refresh window, not a constant as before, but a variable depending on the execution. For this we: remove the synchrony assumption with parameter δ_{sync}, at the cost of only one interaction. Then, we remove the dependency to the bulletin board, and thus to δ_{Board}, in the favorable epochs where no player is decorrupted at the beginning of the Refresh.

Removing synchrony assumption within δ_{sync} After the clock ticks the beginning of a Refresh, every king gathers all TAE.ciphertexts that will be needed to finish the evaluation of the circuit, i.e., not the ones that were already processed through intermediary gates. Then it sends them to its slaves, in the form of a "Refresh" request. Upon receiving this Refresh request message, slaves compute their $\mathsf{ReShare}_k$.contrib on it. This guarantees unicity of the set of ciphertexts on which the slaves of a honest king perform their $\mathsf{ReShare}_k$. However, slaves of a dishonest king may wait indefinitely before receiving a request, and thus keep their old key, which is unsafe. Thus, we specify that they erase their old key as soon as they have received from $t+1$ kings a set of refreshed ciphertexts, which guarantees that, since at least one of them is honest, it will send them to all other kings, who will be able to take over the computation from this point.

Replacing re-publication on the board by distributed update of keys We introduce a new key generation method and an alternative ciphertext refresh mechanism. Finally, note we now take into account the set of secret keys $(r_A)_{j\in A}$ for the PRSS relatively to the current epoch. This time, the steps of a refresh are as follows:

1. Ahead of the *refresh window*, freshly decorrupted player can publish a temporary key to the bulletin board.
2. Players collectively evaluate n instances of the KeyGen protocol of §C.2.5, in an MPC manner, with either the temporary keys or the old ones, once for each recipient player. This creates, for each player j, a new key pair, such that: the private key comes as TAE.ciphertexts, with the signature of $t+1$ players attesting its correctness, which is furthermore privately opened to j, and, such that the public key is publicly opened. Next, players generate new PRSS keys. For this they perform $\binom{n}{n-t}$ executions of the Keygen protocol of §C.2.5. Each execution has parameter a set A of $n-t$ recipient players.
3. Finally, players refresh the other ciphertexts, which are to be used in future epochs, using the subsharing method presented in §4.5.2.

Only then, they can erase from their memory their secret keys of previous epochs, and all plaintexts and ciphertexts related to previous epochs.

References

[AC20] Thomas Attema and Ronald Cramer. "Compressed Σ-Protocol Theory and Practical Application to Plug & Play Secure Algorithmics". In: *CRYPTO*. 2020.

[ACR21] Thomas Attema, Ronald Cramer, and Matthieu Rambaud. "Compressed Σ-Protocols for Bilinear Group Arithmetic Circuits and Application to Logarithmic Transparent Threshold Signatures". In: *to appear in ASIACRYPT*. 2021.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. "Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE". In: *EUROCRYPT*. 2012.

[AJM+21] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. "Reaching Consensus for Asynchronous Distributed Key Generation". In: *PODC*. 2021.

[AJN+16] Prabhanjan Ananth, Aayush Jain, Moni Naor, Amit Sahai, and Eylon Yogev. "Universal Constructions and Robust Combiners for Indistinguishability Obfuscation and Witness Encryption". In: *CRYPTO*. 2016.

[BBCK14] Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. "Asynchronous MPC with a Strict Honest Majority Using Non-equivocation". In: *PODC*. 2014.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Scalable zero knowledge with no trusted setup". In: *Annual international cryptology conference*. Springer. 2019, pp. 701–732.

[BCG21] Elette Boyle, Ran Cohen, and Aarushi Goel. "Breaking the $O(\sqrt(n))$-Bit Barrier: Byzantine Agreement with Polylog Bits Per Party". In: *PODC*. 2021.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. "Semi-homomorphic Encryption and Multiparty Computation". In: *EUROCRYPT*. 2011.

[Bea91] Donald Beaver. "Efficient multiparty protocols using circuit randomization". In: *CRYPTO*. 1991.

[BELO14] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. "How to Withstand Mobile Virus Attacks, Revisited". In: *PODC*. 2014.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. "Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority". In: *CRYPTO*. 2012.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory* (2014).

[BHN08] Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. *Almost-Asynchronous MPC with Faulty Minority*. iacr eprint 2008/416. 2008.

[BHN10] Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. "On the theoretical gap between synchronous and asynchronous MPC protocols". In: *PODC*. 2010.

[BJMS20] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. "Secure MPC: Laziness Leads to GOD". In: *ASIACRYPT*. 2020.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. "Asynchronous Secure Computations with Optimal Resilience (Extended Abstract)". In: *PODC*. 1994.

[Can04] Ran Canetti. "Universally Composable Signature, Certification, and Authentication". In: *CSFW*. 2004.

[Can96] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. 1996.

[CD20] Ignacio Cascudo and Bernardo David. "ALBATROSS: Publicly AttestabLe BATched Randomness Based On Secret Sharing". In: *ASIACRYPT*. 2020.

[CDI05] Ronald Cramer, Ivan Damgaard, and Yuval Ishai. "Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation". In: *TCC*. 2005.

[CDN01] Ronald Cramer, Ivan Damgaard, and Jesper B. Nielsen. "Multiparty Computation from Threshold Homomorphic Encryption". In: *EUROCRYPT*. 2001.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. "Universally Composable Security with Global Setup". In: *TCC*. 2007.

[CFY17] Robert Cunningham, Benjamin Fuller, and Sophia Yakoubov. "Catching MPC Cheaters: Identification and Openability". In: *Information Theoretic Security*. 2017.

[CHP13] Ashish Choudhury, Martin Hirt, and Arpita Patra. "Asynchronous Multiparty Computation with Linear Communication Complexity". In: *DISC*. 2013.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems". In: *ACM CCS*. 2002.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography". In: *J. Cryptol.* (2005).

[CKS11] Jan Camenisch, Stephan Krenn, and Victor Shoup. "A Framework for Practical Universally Composable Zero-Knowledge Protocols". In: *ASIACRYPT*. 2011.

[CLO+13] Ashish Choudhury, Jake Loftus, Emmanuela Orsini, Arpita Patra, and Nigel P. Smart. "Between a Rock and a Hard Place: Interpolating Between MPC and FHE". In: *ASIACRYPT*. 2013.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. "Universally Composable Two-Party and Multi-Party Secure Computation". In: *STOC*. 2002.

[Coh16] Ran Cohen. "Asynchronous Secure Multiparty Computation in Constant Time". In: *PKC*. 2016.

[CS03] Jan Camenisch and Victor Shoup. "Practical Verifiable Encryption and Decryption of Discrete Logarithms". In: *CRYPTO*. 2003.

[CSW19] Ran Cohen, Abhi Shelat, and Daniel Wichs. "Adaptively secure MPC with sublinear communication complexity". In: *Annual International Cryptology Conference*. Springer. 2019, pp. 30–60.

[Dam00] Ivan Damgård. "Efficient Concurrent Zero-Knowledge in the Auxiliary String Model". In: *EUROCRYPT*. 2000.

[DDO+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. "Robust Non-interactive Zero Knowledge". In: *CRYPTO*. 2001.

[DGKN09] I. Damgård, M. Geisler, M. Kroigaard, and J.B. Nielsen. "Asynchronous multiparty computation: Theory and Implementation". In: *PKC*. 2009.

[DHMR08] V. Daza, J. Herranz, P. Morillo, and C. Ràfols. "Ad-Hoc Threshold Broadcast Encryption with Shorter Ciphertexts". In: *Electron. Notes Theor. Comput. Sci.* (2008).

[DLS05] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. "Constant-Round MPC with Fairness and Guarantee of Output Delivery". In: *CRYPTO*. 2005.

[DMR+21] Ivan Damgård, Bernardo Magri, Divya Ravi, Luisa Siniscalchi, and Sophia Yakoubov. "Broadcast-Optimal Two Round MPC with an Honest Majority". In: *CRYPTO*. 2021.

[FO98] Eiichiro Fujisaki and Tatsuaki Okamoto. "A Practical and Provably Secure Scheme for Publicly Verifiable Secret Sharing and Its Applications". In: *EUROCRYPT*. 1998.

[FS01] Pierre-Alain Fouque and Jacques Stern. "One Round Threshold Discrete-Log Key Generation without Private Channels". In: *PKC*. 2001.

[GJPR21] Aarushi Goel, Abhishek Jain, Manoj Prabhakaran, and Rajeev Raghunath. "On Communication Models and Best-Achievable Security in Two-Round MPC". In: *TCC*. 2021.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems". In: *J. ACM* (1991).

[GO07] Jens Groth and Rafail Ostrovsky. "Cryptography in the Multi-string Model". In: *CRYPTO*. 2007.

[GOS12] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "New Techniques for Noninteractive Zero-Knowledge". In: *J. ACM* (2012).

[GPS19a] Yue Guo, Rafael Pass, and Elaine Shi. "Synchronous, with a Chance of Partition Tolerance". In: *CRYPTO*. 2019.

[GPS19b] Yue Guo, Rafael Pass, and Elaine Shi. "Synchronous, with a Chance of Partition Tolerance". In: *CRYPTO*. 2019.

[Gro06] Jens Groth. "Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures". In: *ASIACRYPT*. 2006.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. "Proactive Secret Sharing Or: How to Cope With Perpetual Leakage". In: *CRYPTO*. 1995.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. "Cryptographic Asynchronous Multi-party Computation with Optimal Resilience". In: *EUROCRYPT*. 2005.

[MZW+19] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. "Churp: Dynamic-committee proactive secret sharing". In: *ACM CCS*. 2019.

[NK20] Oded Naor and Idit Keidar. "Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR". In: *DISC*. 2020.

[OY91] Rafail Ostrovsky and Moti Yung. "How to Withstand Mobile Virus Attacks (Extended Abstract)". In: *PODC*. 1991.

[Pas03] Rafael Pass. "On Deniability in the Common Reference String and Random Oracle Model". In: *CRYPTO*. 2003.

[RSY21] Leonid Reyzin, Adam Smith, and Sophia Yakoubov. "Turning HATE into LOVE: Compact Homomorphic Ad Hoc Threshold Encryption for Scalable MPC". In: *Cyber Security Cryptography and Machine Learning*. 2021.

[Sch99] Berry Schoenmakers. "A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting". In: *CRYPTO*. 1999.

[Sha17] Zvika Brakerski and Shai Halevi and Antigoni Polychroniadou. *Four Round Secure Computation without Setup*. 2017.

[SLL10] David Schultz, Barbara Liskov, and Moses Liskov. "MPSS: Mobile Proactive Secret Sharing". In: *ACM Trans. Inf. Syst. Secur.* (2010).

[Sta96] Markus Stadler. "Publicly Verifiable Secret Sharing". In: *EUROCRYPT*. 1996.

[Vin21] Sourav Das and Vinith Krishnan and Irene Miriam Isaac and Ling Ren. *SPURT: Scalable Distributed Randomness Beacon with Transparent Setup*. iacr eprint 2021/100. 2021.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *PODC*. 2019.

[YY01] Adam L. Young and Moti Yung. "A PVSS as Hard as Discrete Log and Shareholder Separability". In: *PKC*. 2001.

[ZSV05] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. "APSS: Proactive secret sharing in asynchronous systems". In: *ACM transactions on information and system security* (2005).

A Complements on the TAE with transparent setup of §3

A.1 Proof of the implementation of TAE in §3.3

A.1.1 Correctness We prove the following more precise proposition, which will be useful to quantify the output of Resize gates (Lemma 4).

Proposition 10. *With respect to the implementations of* TAE *in §3.3, for any $A \leq M$, we have that any homomorphic linear combination of maximum plaintext size $\leq A$ (as defined in 3.2.2), satisfies the definition of being a* well formed ciphertext *of plaintext sizes $\leq A$.*

Lemma 1. *For any two* well formed ciphertext *c and c', with plaintext sizes $\leq A$ and $\leq A'$ such that $A + A' \leq M$, then $c \boxplus c'$ is a* well formed ciphertext *of plaintext sizes $\leq A + A'$, and such that* the *plaintext of $c \boxplus c'$ is the sum of their plaintexts. We have the same result for the $\boxdot\lambda$: the output is of plaintext sizes $\leq \lambda A$.*

Proof. For each $i \in [n]$, the i-th coordinate of $c \boxplus c'$ has plaintext equal to the sum of plaintexts: $s_i + s'_i$, which has size smaller than M. In addition, $s_i + s'_i$ decrypts to $s_i + s'_i \bmod p = (B + B')(i)$. Since $B + B' \in \mathbb{F}_p[X]_t$, this concludes the claim of being a well formed ciphertext of plaintext sizes $\leq A + A'$. Then, the last claim follows from the fact that *the* plaintext of $c \boxplus c'$ is equal to $(B + B')(0) = B(0) + B'(0)$. □

End of the proof of Proposition 10 The proposition follows by applying the following Claim to $a := A$ and to the actual output wire w_{out} of C.

The *Claim* is that for each $a \leq A$: for every fan-out wire w with label $\leq a$, then the output of this wire is a well formed ciphertext c_w, of plaintexts sizes lower than a, and plaintext equal to $\Lambda_{C_w}(m_i)$, where C_w denotes the sub-circuit (prefix) of C consisting of the ancestors of w.

The proof of the Claim is by recurrence on a. Namely, for any $\boxplus$ or $\boxdot$ gate with output wire with label $\leq a$, then, the recurrence assumption applies to the fan-in wire(s), and thus Lemma 1 applies to the gate.

A.1.2 Shares reconstructibility By Proposition 10, any TAE.ciphertext c of m is a well formed ciphertext of m, in particular, the decryptions of the $n = 2t + 1$ coordinates are evaluations of a degree t polynomial $B \in \mathbb{F}_p[X]$ such that $B(0) = m$. Thus, any t of these decryptions along with m, determine the $t + 1$ remaining decryptions. But by constructions, these decryptions are the decryption shares of c, which concludes the proof.

A.1.3 IND-CPA Recall that the first step of **Encrypt** consists in generating a Shamir's secret sharing of the plaintext m. We first recall a classical property of this scheme, then in the next paragraph we deduce IND-CPA from it.

Property 11. *For any fixed plaintext m, for any subset $\mathcal{I}_{\mathcal{A}} \subset [n]$ of t indices, then the distribution obtained by: sample $(b_j)_{j\in[t]}$ uniformly independently, then output $m_i := m + \sum_{j\in[t]} b_j i^j$ for all $i \in \mathcal{I}_{\mathcal{A}}$, is uniform in $\mathbb{F}_p^t$ independent of m.*

The proof follows from invertibility of the Vandermonde determinant.

Deducing IND-CPA We are going to show a bit more than IND-CPA, namely, that no adversary, being allowed to see a set $\mathcal{I}_A \subset [n]$ of t secret keys and make encryption requests, has nonnegligible distinguishing advantage between: the correctly generated $\mathsf{TAE}.\mathbf{Encrypt}$ of a challenge plaintext m of its choice, and, a sample of the following distribution, which we denote $\mathcal{V}$, of elements of $\boldsymbol{\mathcal{C}}$ such that:

- the entries in $\mathcal{I}_A$ are E encryptions of uniform independent values in $\mathbb{F}_p$;
- the entries on the remaining columns are E encryptions of 0.

Since this distribution $\mathcal{V}$ is independent from m, the indistinguishability that we claim indeed implies IND-CPA. Let us prove it by contradiction: let us assume a PPT adversary $\mathcal{A}_{\mathsf{TAE}}$ that has nonneglible advantage in the game above. Then, the following adversary $\mathcal{A}_E$ would have nonneglible advantage in the $(n-t)$-message IND-CPA game of the baseline encryption E. Recall that the $(n-t)$-message IND-CPA game is the variant of the IND-CPA game in which the adversary, instead of sending two challenge plaintexts, sends two sequences of $(n-t)$ challenge plaintexts. A standard argument (with $(n-t)$ hybrid games) shows that indistinguishability of this variant is implied by IND-CPA, with distinguishing advantage at most $(n-t)$ times larger. [Let us make the side remark that [CLO+13, p13] use the 2-message IND-CPA in their proof, in a different context.] We are actually going to show that $\mathcal{A}_E$ wins a slightly stronger game than $(n-t)$-message IND-CPA, i.e., that it will have nonnegligible advantage in distinguishing between the E-encryption of $(n-t)$ messages of its choice, and $(n-t)$ E-ciphertexts of 0. Notice that we also make the variation that the $(n-t)$ messages are encrypted under $n-t$ distinct independently generated public keys, but this can be reduced to the standard version of multi-message IND-CPA.

Let us denote $\mathcal{O}_E$ the oracle challenged by $\mathcal{A}_E$ in the $(n-t)$-message IND-CPA game for E encryption (with the $(n-t)$ keys variation mentionned above). This game starts by $\mathcal{O}_E$ honestly sampling $(n-t)$ public keys for E: $\mathsf{pk}_{t+1}, \ldots, \mathsf{pk}_n$ then giving them to $\mathcal{A}_E$. Then, $\mathcal{A}_E$ runs a copy of $\mathcal{A}_{\mathsf{TAE}}$, and plays the role of the challenging oracle towards $\mathcal{A}_{\mathsf{TAE}}$. Precisely, $\mathcal{A}_E$ samples t public keys pairs: $(\mathsf{sk}_i, \mathsf{pk}_i)_{i\in[1,\ldots,t]}$. Then $\mathcal{A}_E$ re-shuffles the indices of all the $\mathsf{pk}_{i\in[n]}$, of which we recall that she knows the secret keys of t of them, and gives them to $\mathcal{A}_{\mathsf{TAE}}$. Then, upon request of $\mathcal{A}_{\mathsf{TAE}}$ of opening t private keys: if some of them are not in the t set $\mathcal{I}_{\mathcal{A}}$ for which $\mathcal{A}_E$ knows the secret key, then $\mathcal{A}_E$ restarts $\mathcal{A}_{\mathsf{TAE}}$. We claim that this happens with probability $\leq (t+(t+1)/2t+1)^t$. Indeed, all public keys $\mathsf{pk}_{i\in[n]}$ are sampled independently with the same distribution, so $\mathcal{A}_{\mathsf{TAE}}$ has no advantage in distinguishing those for which $\mathcal{A}_E$ knows the secret key.

Provided many such restarts of $\mathcal{A}_{\mathsf{TAE}}$, once the previous step succeeds, denote $\mathcal{I}_{\mathcal{A}}$ the set of corrupt keys, then $\mathcal{A}_E$ continues and honestly responds to the $\mathsf{TAE}.\mathbf{Encrypt}$ requests of $\mathcal{A}_{\mathsf{TAE}}$. Upon receiving one challenge plaintexts m from $\mathcal{A}_{\mathsf{TAE}}$, $\mathcal{A}_E$ computes the first two steps of **Encrypt** on both of them. Concretely, she samples a random polynomial $B = m + \sum_{j\in[t]} b_j X^j$ evaluating to m at 0, then computes the n-sized vector of evaluations $[B(i),\ i \in [n]]$. She then sends the challenge $(n-t)$ plaintext messages: $\{B(i)_{i\in[n]\backslash\mathcal{I}_A}\}$ to $\mathcal{O}_E$. Upon receiving the response ciphertexts $\{c_i,\ i \in [n]\backslash\mathcal{I}_A\}$ from $\mathcal{O}_E$, she then computes the n-sized vector V consisting of:

- The entries in $\mathcal{I}_A$ equal to the correctly generated encryptions $\{E(B(i)),\ i \in \mathcal{I}_A\}$
- The remaining entries are set to the $\{c_i,\ i \in [n]\backslash\mathcal{I}_A\}$.

And sends it to $\mathcal{A}_{\mathsf{TAE}}$ as a response to its challenge. Upon answer a bit b from $\mathcal{A}_{\mathsf{TAE}}$, then $\mathcal{A}_E$ outputs the same bit b to $\mathcal{O}_E$.

Without computing the probabilities, the idea why $\mathcal{A}_E$ has nonnegligible advantage with this strategy is that:

– in the case where the ciphertexts $\{c_i,\ i \in [n]\backslash\mathcal{I}_A\}$ are encryptions of the actual $n-t$ evaluations $\{B(i)_{i\in[n]\backslash\mathcal{I}_A}\}$, then $\mathcal{A}_{\mathsf{TAE}}$ receives from $\mathcal{A}_E$ a correctly generated $\mathsf{TAE}.\mathbf{Encrypt}$ of m.
– in the case where the ciphertexts $\{c_{i,j},\ i \in [n], i \in [n]\backslash\mathcal{I}_A\}$ are encryptions of 0, then, by Property 11 of uniform independence of any t plaintext Shamir shares, we have that what $\mathcal{A}_{\mathsf{TAE}}$ receives from $\mathcal{A}_E$ is undistinguishable from a sample in the distribution $\mathcal{V}$.

A.2 Adding Threshold Private decryption to TAE

The following threshold mechanism **PrivDec** verifiably outputs the plaintext encrypted under the public key pk_r of a designated recipient. It is not used for Theorem 1 in the case where the output of the circuit is meant to be learned by all participants. It will be used in the more advanced proactive protocol of §4.5 , specifically, in the subroutine that generates a new private/public key pair for every player.

Specification

PrivDec. Contrib : $p\mathcal{K} \times s\mathcal{K} \times \boldsymbol{\mathcal{C}} \to \{0,1\}^* \cup \mathsf{abort}$. Notice that the notation $\{0,1\}^*$ is because the length is unspecified, but in our implementation it consists of a E_{pk_r}-ciphertext of an element of $\mathbb{F}_p$.

PrivDec. Combine : $(\{0,1\}^*)^{t+1} \to \{0,1\}^* \cup \mathsf{abort}$ takes $t+1$ outputs of **PrivDec**.Contrib and outputs in $(\{0,1\}^*)$ or abort.

Implementation The following implementation is simple from a conceptual perspective: each player applies **PubDec**.Contrib on the ciphertext, then encrypts the output under the recipient's public key.

PrivDec. Contrib : On input pk_r, which is the recipient's public key, sk_i and c, outputs $E(\mathsf{pk}_r, \mathbf{PubDec}.\mathsf{Contrib}(sk_i, c))$, or abort.

PrivDec. Combine On input $t+1$ correct **PrivDec** shares, decrypt them then apply **PubDec**.Combine.

B Complements on Theorem 1 and UC proof

B.1 Efficient Implementation of the ZK proofs

First note that the non-interactive zero knowledge proof of plaintext knowledge (PoPK) proposed in [BHN10, §5] requires, itself, a trusted setup (to build their key ingredient denoted "ANI-VSS"), although it could be also instantiated by any setup-free NIZK system. In the context of our implementation of TAE, in §3, of (setup-free) TAE encryption, the statements to be proven are naturally expressed in terms of arithmetic relations in (DDH-hard) groups. Hence, instead of compiling these relations into generic arithmetic circuits, there exists a much more efficient way to prove them *directly*, which is provided by the framework [ACR21]. Notice again that this framework however requires a common uniform random string, which is not needed by other NIZK systems in our specific setting of honest majority with $\mathcal{F}_{Board}$.

B.1.1 Example: proof of plaintext knowledge, for Encrypt Thus, although this matter is orthogonal to our main results, for completeness, let us give an explicit example of *succint* NIZK proof of plaintext knowledge (PoPK) from [ACR21] in the case of TAE instantiated with el Gamal in-the-exponent encryption. Before we proceed, let us notice that the case of TAE based on Paillier encryption can then be directly derived. Indeed, the proof below operates on Pedersen commitments, thus it is sufficient to combine it with the proof of [FS01, §3.2] of equality between a Paillier plaintext and a Pedersen commitment.

The prover has a secret plaintext $s \in [0, \dots, p-1] \subset \mathbb{F}_q$. His goal is to compute a $c := \mathbf{Encrypt}(s)$, along with a proof of knowledge of the plaintext of c. Let $(\mathsf{pk}_i)_i$ be the public keys of the n players. The prover samples $B = \sum_{j\in[n]} b_j X^j$ a secret polynomial of degree at most t, such that $b_0 := s$. He computes the $\mathsf{well\ formed\ ciphertext}$ c consisting in the n sized vector of the el Gamal in-the-exponent ciphertexts $c_i = E_{\mathsf{pk}_i}(B(i))$ for all $i \in [n]$. The goal of the prover is to prove knowledge of some degree $\leq t$ polynomial B', such that the $(c_i)_{i\in[n]}$ are encryptions of $B'(i)$.

For each $i \in [n]$, denote r_i the secret random elements of $\mathbb{F}_q$ sampled by the prover to compute c_i. The key point is that the el Gamal ciphertexts c_i are obtained by *linear forms* in the secrets inputs of the prover, namely, the (r_i) and (b_j):

$$E_{\mathsf{pk}_i}(B(i)) = \left\{ r_i.\mathsf{pk}_i \ , \ \Big(\sum_j b_j i^j\Big).g + r_i.g \right\} \ \forall i \in [n] \tag{3}$$

The prover is left with computing a public single compact commitment $P = \textsc{Com}\big((b_j)_{j\in\{0,\dots,t\}}, (r_i)_{i\in[n]}\big)$ and ZK proves that the content of the public P satisfies the n affine forms (3).

Concretely, Com denotes the Pedersen vector commitment, which is a randomized transformation $\mathbb{F}_q \to \mathbb{G}$ where $\mathbb{G}$ (of cardinality q) is the same group as the one used for the el Gamal encryption scheme. Namely, as recalled in [ACR21], this commitment scheme uses as setup several random elements of $\mathbb{G}$. They can be derived from any public uniform random source. Uniformity of the sampling guarantees that these elements, which are automatically generators of $\mathbb{G}$, have statistically no nontrivial linear relation between them. Thus, they enable to commit to a vector of elements of $\mathbb{F}_q$ in a single compact commitment.

To prove that P opens to the n affine forms of (3), the prover can open each of these affine forms by the basic Σ-protocol of [ACR21, §4.1] (which is an easy variation on Chaum-Pedersen), made noninteractive with Fiat-Shamir. Each proof size is $O(n)$: the number of inputs of the affine form. Better: recall that this basic Σ protocol can be *compressed* into a proof of $\log(n)$ size, by the mechanism of [ACR21, §4.1]. Last, recall that the size of opening the n affine forms (3) can be brought down to the one of *one single* affine form, i.e. $O(\log(n))$, by the standard trick of opening a linear combination of them by powers of a random challenge ([ACR21, p. 4.5], [AC20, §5.1]).

B.1.2 Remarks . A direct PoPK (in the univariate case) is described in [Sch99, §3]. However, it is simpler to construct than in our setting, since in his setting, he does not need a blinding factor in the Pedersen commitment to the plaintext, since he assumes that the plaintext is uniformly distributed in $\mathbb{F}_q$. Also, unlike ours, the size is not compressed nor amortized (from $O(n^4)$ down to $O(\log(n))$). The other remark is that there is a hidden difficulty, which is not addressed in [Sch99, §3], and which is, in our setting, that the prover must also provide a range proof that the coefficients b_j of B, are indeed in a small range (in our setting: $[0, \dots, p-1]$), in order to guarantee that decryption of each c_i is tractable. This range proof can be done on the same commitment P, with equally compressed size, using the range proof detailed in [AC20].

B.1.3 Example: proof of correct re-encryption, for PrivDec Denote $(\mathsf{sk}, \mathsf{pk} := g^{\mathsf{sk}})$ the key pair of the prover. His goal is, on input a coordinate c of a well formed ciphertext, encrypted under pk, output a re-encryption c_R of c under the receiver's public key pk_R, and prove that correct computation of this output. By assumption c is an el-Gamal ciphertext of some g^s, for some $s \in [-M, M]$, thus of the form:

$$c = \big((\mathsf{sk}\, r)g, (r+s)g\big) \tag{4}$$

Then, el-Gamal in the exponent decryption returns $s' \in \mathbb{F}_p$ equal to $s \bmod p$, thus of the form

$$s' = s + \lambda p \tag{5}$$

Then, re-encryption of s' is of the form

$$c_R = \big(r'\mathsf{pk}_R, (r'+s')g\big) \tag{6}$$

The prover is left with ZK proving knowledge of $(\mathsf{sk}, s, s', \lambda, r, r')$, such that the affine equations (4), (5), (6) and $(\mathsf{pk} = \mathsf{sk}\, g)$ hold, with respect to the public inputs $(\mathsf{pk}, c, c_R, \mathsf{pk}_R)$. Again, this is an instance of the basic Σ-protocol of [ACR21, §4.1], modulo the hidden extra difficulty of proving the range proof $s' \in [0, \ldots, p-1]$, which is again handled by [AC20].

B.2 Complements on the Triple Generation

We detail in figure 5 the triple generation protocol that we use in the protocol §3.4 proving Theorem 1. The triple generation is an adaptation of [BHN10] our model: TAE (with n public keys locally generated) instead of threshold encryption with trusted setup (with trusted generation of secret keys and of one single public key), and $\mathcal{F}_{ZK}$ (instead of the mechanism of [BHN10, §5.2] with a trusted setup).

We adopt generic notations: $\mathcal{E}$ denotes any threshold encryption scheme that enables (possibly with interactions) the addition of ciphertexts (noted $\boxplus$) and the scalar multiplication (noted $\boxdot$).

B.2.1 Analysis of triple generation In this paragraph we single out guarantees of the triple generation mechanism, that will be used in the UC proof of the protocol §3.4.1. The guarantees and proofs are borrowed unchanged from [BHN10]. In particular, the statements of guarantees 1 and 3 (which hold except with negligible probability) are motivated by the fact that there is no agreement among the players on the multiplication triples (A, B, C), even with respect to one instance of king/slaves, since a dishonest king can send different triples to different players.

Guarantee 1: Let us state this first guarantee in terms of the following game $Triple_{\mathcal{A}}$ between an adversary $\mathcal{A}$ and a challenger defined as:

1. Challenger runs $(pk_i, sk_i) \leftarrow KeyGen() \forall i \in [n]$ and gives all the pk_i to $\mathcal{A}$.
2. $\mathcal{A}$ adaptively outputs a set $\mathcal{S} \subset [n]$ of at most t players, and receives their secret keys. Then he executes once the triple generation protocol $GenTriple_k$ and gives a correct triple (A, B, C) to the challenger.
3. The challenger chooses a random bit $\beta \leftarrow \{0, 1\}$.
   - If $\beta = 0$, it chooses $(a, b) \in R^2$ uniformly and outputs (a,b, ab).
   - If $\beta = 1$, it generates a valid output $Dec(A), Dec(B), Dec(C)$.

Protocol $GenTriple_k$ in the $\mathcal{F}_{ZK}$-hybrid model

Code for slave P_i :

- Upon receiving $(P_k, sid, j, (A_j, B_j, C_j))$, player P_i:
  1. Samples uniformly random plaintexts $u, v \in Z_N$ and compute $U \leftarrow \mathcal{E}(u), V \leftarrow \mathcal{E}(v), X \leftarrow [u \boxdot B_j], Y \leftarrow [v \boxdot A_j]$ and $Z \leftarrow [u \boxdot V]$ and sends $(\text{prove}, sid, (U, u), (V, v), (X, Y, Z))$ to $\mathcal{F}_{ZK}$.
  2. Requests output from $\mathcal{F}_{ZK}$ until receiving $(\text{verification}, sid, 1)$ that proves that : 1) u such that $U \in \mathcal{E}(u)$ and $X \in [u \boxdot B_j]$ 2) v such that $V \in \mathcal{E}(v)$ and $Y \in [v \boxdot A_j]$ 3) u such that $U \in \mathcal{E}(u)$ and $Z \in [u \boxdot V]$.
  3. Sends $(A_j, B_j, C_j, U, V, X, Y, Z)$ to all players.
- Upon receiving $(A_j, B_j, C_j, U, V, X, Y, Z)$ from P_l, player P_i:
  1. Requests the output from $\mathcal{F}_{ZK}$ until receiving $(\text{verification}, sid, , 1)$ for $(A_j, B_j, C_j, U, V, X, Y, Z)$ and computes $A_{j+1} = A_j \boxplus U, B_{j+1} = B_j \boxplus V, C_{j+1} = C_j \boxplus X \boxplus Y \boxplus Z$ and sends back a signature share σ_i on $((A_j, B_j, C_j), (P_k, sid, j, P_l), (A_{j+1}, B_{j+1}, C_{j+1}))$ to P_l. Otherwise do nothing.
- Upon receiving $t + 1$ signature shares σ_l on $((A_j, B_j, C_j), (P_k, sid, j, P_i), (A_{j+1}, B_{j+1}, C_{j+1}))$, computes a signature σ and sends $((A_j, B_j, C_j), (P_k, sid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1}))$.

Code for king P_k :

- Initialize $(A_0, B_0, C_0) = (\mathcal{E}(1, \epsilon), \mathcal{E}(1, \epsilon), \mathcal{E}(1, \epsilon))$ and an empty set $D_{triple,sid}$.
- For $j = 0, ..., t$:
  1. Send $(P_k, sid, j, (A_j, B_j, C_j))$ to all players not in $D_{triple,sid}$.
  2. Upon receiving $((A_j, B_j, C_j), (P_k, sid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1}))$ with a valid signature from a player not in $D_{triple,sid}$, store this answer and add P_i to $D_{triple,sid}$.
- Send $((A_j, B_j, C_j), (P_k, sid, j, P_i, \sigma^{(j)}), (A_{j+1}, B_{j+1}, C_{j+1}))$ for $j = 0, ..., t$ to every player.

Fig. 5: Triple generation protocol

4. $\mathcal{A}$ gets the output and outputs a bit β'.
   The output of the experiment is 1 if $\beta = \beta'(\mathcal{A}$ wins), and 0 otherwise ($\mathcal{A}$ loses).

The first guarantee states that the adversary $\mathcal{A}$ has negligible advantage in this game.

Proof. This follows from the fact that A and B are the result of $t+1$ randomizations, which implies that they were randomized by at least one honest player P_i. Recall that a and b is the sum over $t+1$ randomizers u_j (resp v_j) with one of them being from the honest player P_i. Moreover, every randomizing player P_j proves knowledge of its randomizer u_j (resp v_j). Hence we can, by rewinding, extract the u_j and v_j from the view of the adversary. Thus, distinguishing a and b from uniformly random is equivalent to distinguishing u_i and v_i from uniformly random for at least one honest P_i, which is impossible by the semantic security of the cryptosystem. □

Guarantee 2: We present a slight adaptation of the above guarantee to address the distribution, not only of individual triples for a given *sid* as before, but also of triples between stages. Thus, when a triple (A, B, C) for a stage *sid* is accepted, then we state that the plaintexts of A and B are indistinguishable from uniformly random values which are *statistically independent* from the plaintexts of any triple accepted for any other stage $sid' \neq sid$.

Proof. This follows from the fact that honest players use different randomizers when contributing to different multiplication stages *sid*. □

Guarantee 3: When for the same multiplication stage *sid*, two honest players accept the triples (A, B, C) and (A', B', C'), respectively, then either the plaintexts of (A, B, C) and (A', B', C') are indistinguishable from uniformly random, statistically independent values to the adversary, or the adversary knows the plaintexts of $A - A'$ and $B - B'$.

Proof. This follows from the fact that either there is at least one honest player P_i that has randomized, with the same (u_i, v_i), one triple in some position, but not the other one in another position, or both triples have been randomized by exactly the same set of honest player P_i in exactly the same positions with exactly the same (u_i, v_i). In this case only the adversarially chosen randomizers are different, and they are known to the adversary in the sense that they can be extracted from the adversary in expected polynomial time. □

B.3 Slow King-Dependent Encrypted Randomness Generation

We define a subprotocol $\mathsf{TAE}.\mathbf{Rand}_{\mathbf{slow}\,k}$ that takes no input and allows a king k to output an encrypted value whose plaintext, which we denote "mask" $m \in \mathbb{F}_p$, is uniform unpredictable by the Adversary. We start from the observation that the properties required for the output are actually identical to those of the first two components of the triples, i.e, to be uniform unpredictable. Thus, using a straightforward simplification of the triples generation method of [BHN10], introduced in §2.2 and detailed in Figure 5, we can generate an encrypted random value M from a chain of $t+1$ consecutive randomizations. In details, starting from an initial ciphertext M_0, the king sends a randomization request to all players and waits for the first correct contribution. Upon request, each player P_j randomizes this ciphertext with a randomizer u_j and sends back the result to the king. Iterating this operation $t+1$ times, each time by a different player, we obtain an encrypted random value M.

Remark 3. The previous adaptation of the technique presented in [BHN10] requires $t+1$ consecutive interactions, and is therefore slow. We present in §4.1 a function $\mathsf{TAE}.\mathbf{Rand}_k$ that requires only two consecutive interactions to produce an encrypted random value.

Guarantees: Because we use the same generation technique as for the triples presented in §B.2.1, the same three guarantees remain valid for M, the encryption of a random mask $m \in \mathbb{F}_p$.

B.4 Multiplication protocol, and Analysis

We first recall the protocol *EncBeaver* used in [BHN10] and adapted from [Bea91], which is given in Figure 6, that we use to do multiplications between two values encrypted with a threshold additive scheme. We then formalize and prove the correctness and termination properties of the protocol in Lemma 2 and the privacy property in Lemma 3.

$$EncBeaver(\{X^{(j)}, Y^{(j)}, A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[l]})$$

- We recall that $\boxplus$ denotes the homomorphic addition and $\boxdot$ the homomorphic multiplication by a constant.

1. For each $j \in [l]$, each players P_j computes $F^{(j)} = X^{(j)} \boxplus A^{(j)}$ and $G^{(j)} = Y^{(j)} \boxplus B^{(j)}$.
2. For all $j \in [l]$, the players invoke $PubDec(F^{(j)}, G^{(j)})$ to publicly decrypt $\{f^{(l)}, g^{(l)}\}_{j\in[l]}$.
3. For each $j \in [l]$, the players compute $Z^{(j)} = \mathcal{E}(f^{(j)}g^{(j)}) \boxplus (-f^{(j)} \boxdot B^{(j)}) \boxplus (-g^{(j)} \boxdot A^{(j)}) \boxplus C^{(j)}$ and terminate.

Fig. 6: EncBeaver

Lemma 2. *For every possible $\mathcal{A}$ and for every possible scheduler, protocol EncBeaver achieves:* ***(1)*** Termination: *All the honest players eventually terminate.* ***(2)*** Correctness: *The protocol outputs* $\{\mathcal{E}(x^{(j)}.y^{(j)})\}_{j\in[l]}$.

Proof. Termination: This property follows from the termination property of *PubDec*.

Correctness: This property follows from the fact that for each $j \in [l]$, we have $x^{(j)}y^{(j)} = ((x^{(j)} + a^{(j)}) - a^{(j)})((y^{(i)} + b^{(j)}) - b^{(j)}) = f^{(j)}.g^{(j)} + (-f^{(j)}b^{(j)}) + (-g^{(j)}a^{(j)}) + c^{(j)}$. In particular, we have $\mathcal{E}(x^{(j)}y^{(j)}) = \mathcal{E}(f^{(j)}g^{(j)}) \boxplus (-f^{(j)} \boxdot B^{(j)}) \boxplus (-g^{(j)} \boxdot A^{(j)}) \boxplus C^{(j)}$.

□

Assuming an adversary $\mathcal{A}$ gets valid multiplication triples from P_k (possibly corrupt) and holds two ciphertexts X and Y of x and y respectively, then participating in the multiplication protocol. The following lemma states that the multiplication protocol (recalled in Figure 6) leaks no more information to the adversary than the public values.

Note that we describe the more complex case where the adversary controls the king, as the case of a honest king is much simpler.

Lemma 3 (From [BHN10]). *Let us state this lemma in terms of the following game $Mult_{\mathcal{A}}$ between an adversary $\mathcal{A}$ that controls a king P_k and a challenger defined as:*

1. *Challenger runs* $(pk_i, sk_i) \leftarrow KeyGen()$ *for all* $i \in [n]$ *and gives all the* pk_i *to* $\mathcal{A}$.
2. $\mathcal{A}$ *adaptively outputs a set* $\mathcal{S} \subset [n]$ *(that includes* k*) of at most* t *players, and receives their secret keys. Then he executes the triple generation protocol* $GenTriple_k$*. Recall from Guarantee* 3 *presented in §B.2.1, that honest players* $\{P_j\}_{j\in\mathcal{I}_\mathcal{H}}$ *might respectively accept valid triples* $(A^{(j)}, B^{(j)}, C^{(j)})_j$ *that differ from* $(\delta_a^{(j)}, \delta_b^{(j)}, \delta_c^{(j)})_j$*. In this game, let assume that* $j \in [\mathcal{I}_\mathcal{H}]$*. Finally* $\mathcal{A}$ *gives two encryptions* X *of* x *and* Y *of* y *to the challenger.*
3. *The challenger chooses a random bit* $\beta \leftarrow \{0, 1\}$.
   - *If* $\beta = 0$*, it chooses* $f^{(j_0)}, g^{(j_0)} \in \mathbb{F}_p$ *uniformly for the first* j_0 *and then generates the following* $(f^{(j)}, g^{(j)})$*s each time shifted by* $(\delta_a^{(j)}, \delta_b^{(j)})$,
   - *If* $\beta = 1$*, it successively generates valid decryptions of* $X \boxplus A^{(j)}$, $Y \boxplus B^{(j)}$ *for all* $j \in [\mathcal{I}_\mathcal{H}]$.
4. $\mathcal{A}$ *gets the output and outputs a bit* β'.

The output of the experiment is 1 if $\beta = \beta'$*(*$\mathcal{A}$ *wins), and 0 otherwise (*$\mathcal{A}$ *loses).*

Proof. During the protocol, $\mathcal{A}$ might learn the decryptions $f^{(j)} = x + a^{(j)} \mod n$ and $g^{(j)} = y + b^{(j)} \mod n$ for all $j \in [\mathcal{I}_\mathcal{A}]$. However, $(A^{(j)}, B^{(j)}, C^{(j)})_{j\in[\mathcal{I}_\mathcal{H}]}$ are correct multiplication triples for a multiplication stage *sid*. It follows that either 1) they encrypt values $\{(a^{(j)}, b^{(j)})\}_{j\in[\mathcal{I}_\mathcal{H}]}$ which are uniformly random and independent, or 2) they encrypt values $\{(a^{(j)}, b^{(j)})\}_{j\in[\mathcal{I}_\mathcal{H}]}$ which are individually uniformly random and $(a^{(j+1)}, b^{(j+1)}) = (a^{(j)}, b^{(j)}) + (\delta_a^{(j+1)}, \delta_b^{(j+1)})$ for $(\delta_a^{(j+1)}, \delta_b^{(j+1)})$ known to the adversary.

In the first case, the $\{(f^{(j)}, g^{(j)})\}_{j\in[\mathcal{I}_\mathcal{H}]}$s are uniformly random and independent and thus together leak no information to the adversary. This follows from *Guarantee 1* of §B.2.1.

In the second case, $(f^{(j)}, g^{(j)})$ is uniformly random, and therefore leaks no information to the adversary, and $(f^{(j+1)}, g^{(j+1)}) = (f^{(j)}, g^{(j)}) + (\delta_a^{(j+1)}, \delta_b^{(j+1)})$ and therefore leaks no more information than $(f^{(j)}, g^{(j)})$ to the adversary, as the adversary can compute it from $(f^{(j)}, g^{(j)})$ in expected poly-time.

□

B.5 Resize: interactive reduction modulo p of plaintexts, and Analysis

Resize is a two-steps mechanism that takes as inputs a TAE.ciphertext X of a value $x \in \mathbb{Z}$ of plaintext size below the correct decryption bound A (i.e. all the plaintexts of X are lower than A).. It consumes a TAE.ciphertext M, of a random value $m \in \mathbb{F}_p$, generated from TAE.$\mathbf{Rand}_{\mathbf{slow}\,k}$ as introduced in §B.3, denoted the "mask". It outputs a TAE.ciphertext of $x' = x \bmod p$ of small plaintext sizes: $|x'| \leq (n+1)(p-1)$. Before we proceed, let us mention that the above claim, made precise in (2) of the Lemma below, is formulated in terms the sizes of the plaintexts of the n entries of x', which is specific to our implementation of TAE. But the claim actually holds more generally, namely, it holds that, for any TAE, then the output of Resize is a TAE.ciphertext of plaintext size $\leq (n+1)(p-1)$. In details, Resize consists of:

1. a masking operation between X and M to produce $X \boxplus ((-1) \boxdot M)$, which is then opened using **PubDec** to recover the plaintext value $x - m$,
2. then king computes the encryption $X' :=$ TAE.$\mathbf{Encrypt}(x - m, \mathsf{pk}) \boxplus M$ in order to remove the mask M.

We now proof the correctness and termination of Resize in Lemma 4 and the privacy property in Lemma 5.

Lemma 4. *For every possible $\mathcal{A}$, for every possible scheduler and for an honest king P_k, protocol* Resize *achieves:* ***(1)*** Termination$_k$: *All the honest players eventually terminate.* ***(2)*** Correctness$_k$: *The protocol outputs a* well formed ciphertext *X' of x, such with plaintexts sizes $\leq (n+1)(p-1)$.*

Proof. Termination$_k$: This property follows from the fact that at least $t+1$ players are honest.

Correctness$_k$: This property follows from the correctness properties of our TAE (Proposition 10, in particular Lemma 1). □

Lemma 5 (Adapted from Lemma 3). *Provided we can write a game $Res_{\mathcal{A}}$ similar to the one presented in Lemma 3, we can state that the* Resize *protocol leaks no information to the adversary.*

Proof. This follows directly from the proof of Lemma 3 applied on $x+m$ instead on $x+a$ (resp $y+b$). □

B.6 UC proof of the protocol of §3.4 for Theorem 1

B.6.1 Roadmap We prove that the protocol implements, in the sense of uniform composability (UC), the ideal functionality $\mathcal{F}_C$ of secure evaluation of an arithmetic circuit C over $\mathbb{F}_p$. Concretely, we exhibit a simulator $\mathcal{S}\text{im}$ such that no non-uniform PPT environment $\mathcal{Z}$, which can choose the honest inputs, observe the honest outputs, and fully controls t out of the $n = 2t+1$ players (via an adversary $\mathcal{A}$), can distinguish between: (i) the real execution $REAL_{\Pi}$ of the protocol, where honest players interact with the corrupt ones, and (ii) the ideal execution $IDEAL_{\mathcal{F}_C}$ where: honest players interact with an ideal functionality $\mathcal{F}$, and the simulator $\mathcal{S}\text{im}$ interacts, on the one side, with $\mathcal{F}_C$ on behalf of corrupt players, and on the other side with the corrupt players: both on behalf of both honest players (to be sure, in the UC definition $\mathcal{S}\text{im}$ is instead allowed to simulate the corrupt players to $\mathcal{A}$, but we will not need this additional power) and of the ideal functionality $\mathcal{F}_{ZK}$. However we do not allow $\mathcal{S}\text{im}$ to *use the witnesses* given by corrupt players to the functionalities $\mathcal{F}_{ZK}^{1:M}$ and $\mathcal{F}_{ZK}$ simulated by $\mathcal{S}\text{im}$, i.e., *we do not allow rewinding.*

To keep the focus on the core security properties, we consider in the simulation, similarly to [BHN10], *deterministic* functions. However, we observe that the simulation can be easily modified to cover any probabilistic polynomial-time functions, by letting the environment $\mathcal{Z}$ choose the random input r_i for random gates.

Finally, we consider that the protocol instructs players to initialize $\mathcal{F}_{\text{Board}}$ at the beginning of every execution (parametrised by the "session-id" (sid)). This model, known as *local setup*, is also assumed in the proof of [BHN08]. It enables $\mathcal{S}\text{im}$ to simulate $\mathcal{F}_{\text{Board}}$, and thus provide fake keys to the adversary on behalf of honest players. In §B.6.2, we detail the construction of our simulator and in §B.6.2 we prove the overall protocol. We then discuss the global setup model in §B.7.

B.6.2 Description of the simulator Let us now exhibit $\mathcal{S}\text{im}$, the proof of undistinguishability is provided in §B.6.2. The main difference with [BHN10] (and also [CDN01; Coh16]) is that, there, $\mathcal{S}\text{im}$ had the additional power to simulate the trusted setup, and therefore learn (even choose) the secret keys assigned to corrupt players. He could therefore compute the decryption shares of corrupt players, then use this information to simulate compatible honest decryption shares. Our main innovation compared to [BHN10] is a technique to enable $\mathcal{S}\text{im}$ to still simulate the opening of any ciphertext. We exhibit this technique below in the simulation of opening (4). Concretely, $\mathcal{S}\text{im}$ uses the shares reconstructibility (§3.2.4) of TAE to infer corrupt decryption shares, from the knowledge of the $t+1$ decryption shares of the simulated honest players. We denote $\mathcal{I}_{\mathcal{A}}$ the t indices of corrupt players and $\mathcal{I}_{\mathcal{H}}$ the $t+1$ indices of honest ones.

(0) *Simulating the setup.* $\mathcal{S}$im starts by simulating $\mathcal{F}_{\text{Board}}$ and collects the public keys pk_i of corrupt players ($i \in \mathcal{I}_{\mathcal{A}}$) on behalf of $\mathcal{F}_{\text{Board}}$. For every honest player ($i \in \mathcal{I}_{\mathcal{H}}$), it correctly KeyGens a "fake" key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$, and defines $\mathbf{pk} = (pk_1, ..., pk_n)$. Note that non-received keys are set to $\perp$. Upon read requests from corrupt players to $\mathcal{F}_{\text{Board}}$, it returns the previous $\mathsf{pk}_{i \in \mathcal{I}_{\mathcal{A}} \cup \mathcal{I}_{\mathcal{H}}}$.

(1) *Simulating the Input distribution.* $\mathcal{S}$im simulates the operations of all honest players in the input distribution phase. $\mathcal{S}$im sets every ciphertext of a simulated honest player to 0, i.e $\widetilde{x_i} := 0$ for $i \in \mathcal{I}_{\mathcal{H}}$. Then $\mathcal{S}$im simulates honestly $\mathcal{F}_{ZK}^{1:M}$, *without rewinding*, i.e., it immediately erases from its memory the secret witnesses input from the adversary, upon having checked correctness (or not) of these inputs. When the adversary sends a request to $\mathcal{F}_{ZK}^{1:M}$ for $j \in \mathcal{I}_{\mathcal{H}}$ on behalf of a corrupted player, $\mathcal{S}$im responds with a confirmation of the validity of the ciphertext $\widetilde{c}_j$ and immediately deletes the plaintexts, thus preventing itself from subsequently using them beyond the ciphertexts $c_{i \in \mathcal{I}_{\mathcal{A}}}$. From these ciphertexts, $\mathcal{S}$im can recover the plaintexts $x_{i \in \mathcal{I}_{\mathcal{A}}}$ that it just erased, by using the $t+1$ decryption keys of simulated honest players to apply threshold decryption. Likewise, $\mathcal{S}$im immediately erases the secret keys that it receives from corrupt players on behalf of $\mathcal{F}_{ZK}^{1:M}$.

(2) - (3) *Simulating the Computation.* $\mathcal{S}$im honestly simulates $\mathcal{F}_{ZK}$ throughout the computation phase. $\mathcal{S}$im honestly follows the protocol, on behalf of honest players, using the fake inputs. Specifically, $\mathcal{S}$im runs the protocol honestly for additions, possibly with Resize as detailed in §B.5. Multiplications are evaluated using the Beaver [Bea91] technique recalled in Figure 6. $\mathcal{S}$im runs the protocol honestly for multiplications and Resize and, as before, aborts if some share from a corrupted player is not correct.

(4) *Simulating the threshold decryption.* Now the kings (both corrupt and simulated) who followed the protocol so far have formed ciphertexts $c_{\widetilde{y}}^{(k)}$ of the evaluation $\widetilde{y}$. $\mathcal{S}$im handles the decrypted inputs $x_{i \in \mathcal{I}_{\mathcal{A}}}$ of corrupt players to $\mathcal{F}_C$ and retrieves from it an evaluation of the circuit on the *actual inputs*: $y := C(x_{i \in \mathcal{I}_{\mathcal{A}}}, x_{i \in \mathcal{I}_{\mathcal{H}}})$ Since honest players also receive y from $\mathcal{F}_C$ and output it, $\mathcal{S}$im must thus simulate the *false* opening of the $c_{\widetilde{y}}^{(k)}$ into y. In what follows we describe the strategy for this simulation, which we denote a *simulated decryption of* $c_{\widetilde{y}}^{(k)}$ to y. For honest kings k, $\mathcal{S}$im simply returns y to the corrupt players on behalf of k. For corrupt kings, $\mathcal{S}$im will use *twice* the shares reconstruction, in a nonstandard way as follows:

   (a) First it does a *corrupt share inference*: from the $t+1$ decryption shares $\widetilde{d}_{i \in \mathcal{I}_H}$ of $c_{\widetilde{y}}^{(k)}$ held by simulated honest players, it applies $\mathsf{ShInfer}$ to deduce the t corrupt decryption shares $\widetilde{d}_{i \in \mathcal{I}_A}$ of $c_{\widetilde{y}}^{(k)}$.

   (b) Second to simulate consistent decryption shares from these $\widetilde{d}_{i \in \mathcal{I}_A}$ and the plaintext y, it applies ShReco to deduce the (unique) corresponding $t+1$ honest decryption shares $d_{i \in \mathcal{I}_H}$ of y.

   It sends them to k on behalf of honest players, along with *fake* proofs that these are correct decryption shares of $c_{\widetilde{y}}^{(k)}$, namely, it sends $\mathsf{verification}$ to k on behalf of $\mathcal{F}_{ZK}$.

B.6.3 Proof of undistinguishability with a real execution We go through a series of hybrid games that will be used to prove the indistinguishability of the real and ideal worlds. The output of each game is the output of the environment.

The Game $REAL_{\Pi,\mathcal{A},\mathcal{Z}}$ This is exactly the execution of the protocol Π in the real-model with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ (and ideal functionalities ($\mathcal{F}_{\text{Board}}$, $\mathcal{F}_{ZK}^{1:M}$, $\mathcal{F}_{ZK}$)).

The Game $Hyb^1_{\Pi,\mathcal{A},\mathcal{Z}}$ In this game, we modify the real-model experiment in the setup and input distribution steps. The keys of honest players published in $\mathcal{F}_{Board}$ are replaced by honestly generated keys. Moreover, the secret keys and plaintext inputs of corrupt players are now extracted from the non-interactive ZK proofs through $\mathcal{F}_{ZK}^{1:M}$.

Claim. $\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \text{Hyb}^1_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. In both cases, the distribution of $t+1$ keys the adversary receives is chosen uniformly at random in $p\mathcal{K}$. Moreover for the adversary, the outputs of $\mathcal{F}_{ZK}^{1:M}$ are the same in both games. □

From this point, the shares of corrupt players will be infered by using the extracted keys.

The Game $Hyb^2_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\text{Hyb}^1_{\Pi,\mathcal{A},\mathcal{Z}}$ except for the computation of the decryption shares of $c_y^{(k)}$ from honest players during the computation, which are now simulated. Specifically, the decryption shares of corrupt players of the result are infered thanks to the corrupt secret keys extracted from $\mathcal{F}_{ZK}^{1:M}$, then we deduce the output y from the ideal functionality $\mathbb{F}_C$, then use ShReco with input the corrupt shares and y, which outputs compatible honest decryption shares of y.

Claim. $\text{Hyb}^1_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \text{Hyb}^2_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. We replace the decryption shares of $c_y^{(k)}$ of honest players by simulated decryption shares obtained from ShReco. By the definition of ShReco detailed in §3.2.4, their are identical to what honest players output.

□

Recall that to multiply two ciphertexts X and X', we use Beaver triples [Bea91] A, B and C and publicly decrypt intermediary values $F = X \boxplus A$ and $G = X' \boxplus B$. Moreover, remember that, in order to reduce interactions, we use a Resize protocol that follows the same a similar technique with $Y = X \boxplus M$. Finally, recall from *Guarantee* 3 presented in §B.2.1, that honest players $\{P_j\}_{j\in\mathcal{I}_\mathcal{H}}$ might respectively accept valid triples $(A^{(j)}, B^{(j)}, C^{(j)})_j$ (resp masks $\{M^{(j)}\}_j$) that differ from $(\delta_a^{(j)}, \delta_b^{(j)}, \delta_c^{(j)})_j$ (resp $\delta_m^{(j)}$). Thus, because it involves partial decryptions, we introduce the following sub-games, $\text{Hyb}^3_{\Pi,\mathcal{A},\mathcal{Z}}$ and $\text{Hyb}^4_{\Pi,\mathcal{A},\mathcal{Z}}$.

The Game $Hyb^3_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\text{Hyb}^2_{\Pi,\mathcal{A},\mathcal{Z}}$ except for the computation of the decryption shares of $F^{(j)}$ and $G^{(j)}$ (resp $Y^{(j)}$ for Resize) from honest players during a multiplication (resp Resize) where we do a simulated decryption $F^{(j)}$ and $G^{(j)}$ (resp $Y^{(j)}$) to the public decryption of $F^{(j)}$ and $G^{(j)}$ (resp $Y^{(j)}$) instead of the real ones.

Claim. $\text{Hyb}^2_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \text{Hyb}^3_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. Recall that we are using the correct inputs $F^{(j)}$ (resp $G^{(j)}, Y^{(j)}$). Therefore, except with negligible probability, $F^{(j)}$ (resp $G^{(j)}, Y^{(j)}$) contains the value $f^{(j)}$ (resp $g^{(j)}, y^{(j)}$) returned by the ideal functionality and the simulated output is indistinguishable from the honest decryption of $F^{(j)}$ (resp $G^{(j)}, Y^{(j)}$). □

The Game $Hyb^4_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\mathrm{Hyb}^3_{\Pi,\mathcal{A},\mathcal{Z}}$ except for for the computation of the decryption shares of $F^{(j)}$ and $G^{(j)}$ (resp $Y^{(j)}$) from honest players during a multiplication (resp Resize) where we do a simulated decryption of $F^{(j)}$ and $G^{(j)}$ (resp $Y^{(j)}$) to two random elements $f^{(j)}, g^{(j)} \in \mathbb{F}_p$ (resp $y^{(j)} \in \mathbb{F}_p$).

Claim. $\mathrm{Hyb}^3_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \mathrm{Hyb}^4_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. This follows from lemma 3 (resp Lemma 5) in which we argued that the multiplication protocol (resp Resize) leaks no information to the adversary. □

The Game $Hyb^5_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\mathrm{Hyb}^4_{\Pi,\mathcal{A},\mathcal{Z}}$ except in the input distribution stage. The ciphertexts from every honest player P_i contain encryptions of 0 rather than x_i.

Claim. $\mathrm{Hyb}^4_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \mathrm{Hyb}^5_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. This follows from the IND-CPA security of TAE. □

Note that up to this point, in all the previous games that we described, we still have *i*) all corrupt share inferences are done thanks to the corrupt secret keys extracted from $\mathcal{F}_{ZK}^{1:M}$ and *ii*) the corrupt plaintexts are recovered thanks to $\mathcal{F}_{ZK}^{1:M}$.

The Game $Hyb^6_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\mathrm{Hyb}^5_{\Pi,\mathcal{A},\mathcal{Z}}$ except that *i*) now to make the corrupt share inference, we do not use the extracted key for corrupt players but instead we apply the technique detailed in (4).(*a*) (ShInfer) to infer the decryption shares from corrupt players by using the $t+1$ decryption shares of simulated honest players.

Claim. $\mathrm{Hyb}^5_{\Pi,\mathcal{A},\mathcal{Z}} \equiv \mathrm{Hyb}^6_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. The view of the adversary $\mathcal{A}$ does not change regardless of the (equivalent) technique used to (deterministically) infer the corrupt decryption shares. □

Claim. $\mathrm{Hyb}^6_{\Pi,\mathcal{A},\mathcal{Z}} = \mathrm{IDEAL}_{f,\mathcal{A},\mathcal{Z}}$

Proof. This follows since the behaviour of ideal functionalities ($\mathcal{F}_{Board}$, $\mathcal{F}_{ZK}^{1:M}$), the behaviour of the simulated ideal functionality $\mathcal{F}_{ZK}$ (which we recall differs from the actual $\mathcal{F}_{ZK}$ only at the final opening stage, where it provides a false proof of correct decryption share) and the behaviour of the honest parties in $\mathrm{Hyb}^6_{\Pi,\mathcal{A},\mathcal{Z}}$ is identical to the simulation done by $\mathcal{S}$im. □

B.7 Moving Theorem 1 to a global setup

The proof of our protocol in §3.4 leverages the fact that $\mathcal{F}_{\mathrm{Board}}$ is initialized at the beginning of the execution, and thus that it can be simulated by $\mathcal{S}$im. This is what enables $\mathcal{S}$im to generates fake keys on behalf of honest players, and thus decrypt the ciphertexts encrypted by the adversary with these keys (just as in [BHN08, B]). It is possible to remove this assumption, but at the price of allowing the simulator to "rewind" the adversary. Concretely, we allow $\mathcal{S}$im to *use* the inputs that corrupt players give to the functionalities that $\mathcal{S}$im simulates. Namely, the witnesses given by the adversary to $\mathcal{F}_{ZK}$, and also the plaintext inputs broadcast along with a NIZK. Notice that, more precisely, $\mathcal{S}$im needs to extract all decryption shares. It is indeed able to do it, since these

shares can be deduced from the witnesses in the the explicit relations that we specify above. So this makes non black box use of our implementation of TAE. Alternatively, we could have specified in the protocol that decryption shares should be extractable from the ZK proofs sent.

Of independent interest, one could possibly *also* like to implement $\mathcal{F}_{ZK}$ with a global setup. Concretely, instead of obtaining the random string by a call to a random beacon, have it instead as a fixed public parameter, known a priori by the Environment. For instance, the decimals of Pi, or a 2009 NYT cover (as in Bitcoin). [Pas03, p18-19] achieves it in two rounds, provided a simulator with access to the RO queries of the adversary. [CDPW07] achieves it from a key registration that requires players to prove *knowledge* of a secret key (the "$\mathcal{F}_{KRK}$"). However, strong impossibility results for ZK under global setup are stated by [Pas03; CDPW07]. Fortunately, lighter primitives than $\mathcal{F}_{ZK}$ are sufficient for MPC (as the ones of [CKS11]).

C Detailed ingredients of theorem 2

C.1 New Constant time triple generation

C.1.1 The Preprocessing phase protocol We first present the almost-asynchronous preprocessing phase protocol *PreProc*, which is given in Figure 7. We then formalize the properties of the protocol in lemma 6. We later prove it in §C.1.3, thanks to two subprotocols, *TripExt* that we details in §C.1.2 and *EncBeaver*, detailed in §B.4.

Protocol *PreProc*

First synchronous broadcast round

1. **Triple distribution** - For $i \in [n]$, every player P_i executes the following code:
   - Act as a dealer D and broadcast a random multiplication triples $\{(X^{(i)}, Y^{(i)}, Z^{(i)}\}$.

The remaining asynchronous protocol

2. **Triple verification** - For $i \in [n]$, every player P_i executes the following code:
   - The players verify R_{trip} for all triples $(\{(X^{(i)}, Y^{(i)}, Z^{(i)}\}_{i\in[n]})$ and output a set $\mathcal{U}$ consisting of $t + 1 + t'$ players who broadcast correct triples.
3. **Triple extraction and termination**- The players execute the following code:
   - The players execute $TripExt(\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in\mathcal{U}})$, output $\frac{t+t'}{2} + 1 - t'$ triples $\{(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C})_i\}$ for $i \in [\frac{t+t'}{2} + 1 - t']$ and terminate.

Fig. 7: Preprocessing overview

Lemma 6. *For every possible $\mathcal{A}$ and every possible scheduler, protocol PreProc achieves:* ***(1)*** Termination: *All honest players terminate the protocol.* ***(2)*** Correctness: *The $\frac{t+t'}{2} + 1 - t'$ output triples will be multiplication triples.* ***(3)*** Privacy: *The $\frac{t+t'}{2} + 1 - t'$ output triples are random and unknown to $\mathcal{A}$*

C.1.2 Triples Extraction The idea is to interpolate three polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ from the broadcasted triples and use them to produce new values. Our protocol is adapted from the protocol for the transformation of t-shared triples proposed in [CHP13]. The main difference is that we don't consider shares, but we work instead on values encrypted using a threshold additive homomorphic encryption scheme. This enables all players in an instance led by a king P_k to run the same protocol with the same inputs and produce the same outputs.

In greater detail, protocol $TripExt$ takes as input $t+1+t'$ *correct* triples, say $\{(A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[t+1+t']}$, where $A^{(j)} = \mathcal{E}(a^{(j)}), B^{(j)} = \mathcal{E}(b^{(j)})$ and $C^{(j)} = \mathcal{E}(c^{(j)})$ and where, for all j, it holds that $c^{(j)} = a^{(j)}.b^{(j)}$. Note the here t' denotes the number of *correct* triples broadcasted by $\mathcal{A}$. $TripExt$ then produces $t+1+t'$ triples, say $\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']}$ [5], such that the following holds:

(1) there exist polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ of degree at most $\frac{t+t'}{2}$, $\frac{t+t'}{2}$ and $t+t'$ respectively, such that $\mathbb{x}(i) = x^{(i)}, \mathbb{y}(i) = y^{(i)}$ and $\mathbb{z}(i) = z^{(i)}$ holds for $i \in [t+1+t']$.

(2) The ith output triple $(X^{(i)}, Y^{(i)}, Z^{(i)})$ is a multiplication triple iif the ith input triple $(A^{(i)}, B^{(i)}, C^{(i)})$ is a multiplication triple.

(3) If $\mathcal{A}$ knows t' input triples and if $t' \leq \frac{t+t'}{2}$, then he learns t' distinct values of $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$, implying $\frac{t+t'}{2}+1-t'$ degrees of freedom, i.e remaining independent distinct values of $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ that would be needed to uniquely determine these polynomials.

The core functionality of this protocol that enables to build the three polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ is inherited from the verification of the multiplication triples from [BFO12]. Specifically, the two polynomials $\mathbb{x}$ and $\mathbb{y}$ are entirely defined by the first and second components $(a^{(i)}, b^{(i)})$ of the first $\frac{t+t'}{2}+1$ triples. The construction of $\mathbb{z}(.)$ is not as straightforward due to the difference in degree. We use $\mathbb{x}(.)$ and $\mathbb{y}(.)$ to compute $\frac{t+t'}{2}$ "new points" and use the remaining $\frac{t+t'}{2}$ available triples $(A^{(i)}, B^{(i)}, C^{(i)})_{i\in[\frac{t+t'}{2}+2, t+1+t']}$ to compute their products . Ultimately, $\mathbb{z}(.)$ is both defined by the last components of the first $\frac{t+t'}{2}+1$ triples and by the $\frac{t+t'}{2}$ computed products.

Finally, the random outputs, unknown to $\mathcal{A}$, are then extracted as $\{(\mathbb{X}(\beta_j), \mathbb{Y}(\beta_j), \mathbb{Z}(\beta_j))\}_{j\in[\frac{t+t'}{2}+1-t']}$.

Lemma 7. *Let $\{(A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[t+1+t']}$ be a set of $t+1+t'$ broadcast triples. Then for every possible adversary $\mathcal{A}$ and every possible scheduler, protocol $TripExt$ achieves:* ***(1)*** Termination: *All the honest players eventually terminate the protocol* ***(2)*** Correctness: *The protocol outputs $\frac{t+t'}{2}+1-t'$ triples ($\boldsymbol{A}_i = \mathbb{X}(\beta_i)$,$\boldsymbol{B}_i = \mathbb{Y}(\beta_i)$ and $\boldsymbol{C}_i = \mathbb{Z}(\beta_i)$) for $i \in [\frac{t+t'}{2}+1-t']$ such that the following holds* ***(a)*** *There exist polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ of degree $\frac{t+t'}{2}$, $\frac{t+t'}{2}$ and $t+t'$ respectively. With $\mathbb{X}(i) = \mathcal{E}(\mathbb{x}(i)$ for $i \in [t+1+t']$ (resp $\mathbb{Y}, \mathbb{Z}$), it holds: $\mathbb{X}(i) = X^{(i)}, \mathbb{Y}(i) = Y^{(i)}$ and $\mathbb{Z}(i) = Z^{(i)}$.* (**b**) *$\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$ holds iff all the input triples are multiplication triples.* ***(3)*** Privacy: *The view of $\mathcal{A}$ in the protocol is distributed independently of the output multiplication triples $\{(\boldsymbol{A}_i = \mathbb{X}(\beta_i)$, $\boldsymbol{B}_i = \mathbb{Y}(\beta_i)$, $\boldsymbol{C}_i = \mathbb{Z}(\beta_i))\}$ for $i \in [\frac{t+t'}{2}+1-t']$.*

Proof. Termination: This property follows from the termination property of $EncBeaver$ (see Lemma 2).

Correctness: By construction, it is ensured that the polynomials $\mathbb{x}, \mathbb{y}$ and $\mathbb{z}$ are of degree $\frac{t+t'}{2}$, $\frac{t+t'}{2}$ and $t+t'$ respectively and $\mathbb{X}(i) = X^{(i)}, \mathbb{Y}(i) = Y^{(i)}$ and $\mathbb{Z}(i) = Z^{(i)}$ holds for $i \in [t+1+t']$. To

[5] Following our notations, $X^{(j)} = \mathcal{E}(x^{(j)}), Y^{(j)} = \mathcal{E}(y^{(j)})$ and $Z^{(j)} = \mathcal{E}(z^{(j)})$

Protocol $TripExt(\{(A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[t+1+t']})$

1. For each $j \in [\frac{t+t'}{2} + 1]$, the players locally set $X^{(j)} = A^{(j)}$, $Y^{(j)} = B^{(j)}$, and $Z^{(j)} = C^{(j)}$.
2. Let the points $\{j, x^{(j)}\}_{j\in[\frac{t+t'}{2}+1]}$ and the points $\{j, y^{(j)}\}_{j\in[\frac{t+t'}{2}+1]}$ define the polynomials $\mathbb{x}(.)$ and $\mathbb{y}(.)$ respectively of degree at most $(\frac{t+t'}{2})$.
3. The players compute $X^{(j)} = \mathbb{X}(\alpha_j)$ and $Y^{(j)} = \mathbb{y}(\alpha_j)$ for each $j \in [\frac{t+t'}{2} + 2, t + 1 + t']$. Computing a new point on a polynomial of degree $\frac{t+t'}{2}$ is a linear function of $\frac{t+t'}{2} + 1$ given unique points on the same polynomial.
4. The players execute $EncBeaver(\{X^{(j)}, Y^{(j)}, A^{(j)}, B^{(j)}, C^{(j)}\}_{j\in[\frac{t+t'}{2}+2,t+1+t']}$ to compute $\frac{t+t'}{2}$ values $\{Z^{(j)}\}_{j\in[\frac{t+t'}{2}+2,t+1+t']}$. Let the points $\{j, z^{(j)}\}_{j\in[t+1+t']}$ define the polynomial $\mathbb{z}(.)$ of degree at most $t + t'$.
5. The players compute $\mathbf{A}_i = \mathbb{X}(\beta_i)$,$\mathbf{B}_i = \mathbb{Y}(\beta_i)$ and $\mathbf{C}_i = \mathbb{Z}(\beta_i)$ for $i \in [\frac{t+t'}{2} + 1 - t']$ and terminate.

Fig. 8: Triple extraction

argue the second statement in the correctness property, we first show that if the input triples are multiplication triple then $\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$ holds. For this, it is enough to show the multiplicative relation $\mathcal{E}(\mathbb{z}(i)) = \mathcal{E}(\mathbb{x}(i)\mathbb{y}(i))$ holds for $i \in [t + 1 + t']$. For $i \in [\frac{t+t'}{2} + 1]$, the relation holds since we have $X^{(i)} = A^{(i)}, Y^{(i)} = B^{(i)}, Z^{(i)} = C^{(i)}$ and the triple $(A^{(i)}, B^{(i)}, C^{(i)})$ is a multiplication triple by assumption. For $i \in [\frac{t+t'}{2} + 2, t + t' + 1]$, we have $\mathcal{E}(\mathbb{z}(i)) = \mathcal{E}(\mathbb{x}(i)\mathbb{y}(i))$ due to the correctness of the protocol *EncBeaver* and the assumption that the triples used in *EncBeaver*, namely $\{(A^{(i)}, B^{(i)}, C^{(i)})\}_{i\in[\frac{t+t'}{2}+2,t+1+t']}$ are multiplication triples. Proving the other way, that is, if $\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$ is true then all the input triples are multiplication triples is easy. Since $\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$, it implies that $\mathcal{E}(\mathbb{z}(i)) = \mathcal{E}(\mathbb{x}(i)\mathbb{y}(i))$ for $i \in [t + 1 + t']$. This trivially implies $\{(A^{(i)}, B^{(i)}, C^{(i)})\}_{i\in[\frac{t+t'}{2}]}$ are multiplication triples. On the other hand, if some triple in $\{(A^{(i)}, B^{(i)}, C^{(i)})\}_{i\in[\frac{t+t'}{2}+1,t+1+t']}$, say $(A^{(j)}, B^{(j)}, C^{(j)})$ is not a multiplication triple, then $(X^{(j)}, Y^{(j)}, Z^{(j)})$ is not a multiplication triple as well (by the correctness of the Beaver's technique), which is a contradiction.

Thus, the triples ($\mathbf{A}_i$, $\mathbf{B}_i$, $\mathbf{C}_i$), defined as ($\mathbf{A}_i = \mathbb{X}(\beta_i)$,$\mathbf{B}_i = \mathbb{Y}(\beta_i)$ and $\mathbf{C}_i = \mathbb{Z}(\beta_i)$), are valid multiplication triples for $i \in [\frac{t+t'}{2} + 1 - t']$.

Privacy: We show that the view of the adversary $\mathcal{A}$ in the protocol $TripExt$ is distributed independently of the multiplication triples ($\mathbf{A}_i$, $\mathbf{B}_i$, $\mathbf{C}_i$). In other words, for $\mathcal{A}$ all possible multiplication triples output by $TripExt$ are equiprobable.

We first note that, $\mathcal{A}$ learns at most $t' \leq \frac{t+t'}{2}$ points on the polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$. To assert this, we first show that if $\mathcal{A}$ knows more than $\frac{t+t'}{2}$ input triples, then it knows all the three polynomials completely. Second, we show that if $\mathcal{A}$ knows the un-encrypted input triple $(a^{(i)}, b^{(i)}, c^{(i)})$, then it also knows the un-encrypted output triple $(x^{(i)}, y^{(i)}, z^{(i)})$. If $i \in [\frac{t+t'}{2} + 1]$, this follows trivially since $(x^{(i)}, y^{(i)}, z^{(i)})$ is the same as $(a^{(i)}, b^{(i)}, c^{(i)})$. Else if $i \in [\frac{t+t'}{2} + 2, t + 1 + t']$, then $\mathcal{A}$ knows the triple $(a^{(i)}, b^{(i)}, c^{(i)})$ which is used to compute $Z^{(i)}$ from $X^{(i)}$ and $Y^{(i)}$. Since the

values $(x^{(i)} + a^{(i)})$ and $(y^{(i)} + b^{(i)})$ are disclosed during the computation of $Z^{(i)}$, $\mathcal{A}$ knows $x^{(i)}, y^{(i)}$ and hence $z^{(i)}$[6].

Second, we note that, since degree of $\mathbb{x}$ is at most $\frac{t+t'}{2}$, for all choice of $\{A_j\}_{j\in[\frac{t-t'}{2}+1]}$ there exist a unique polynomial $\mathbb{x}(.)$ of degree at most $\frac{t+t'}{2}$ which will be consistent with these points $(\mathbb{X}(\gamma_j) = \mathbf{A}_j)_{j\in[\frac{t-t'}{2}+1]}$ and with the prior knowledge of $\mathcal{A}$. Thus, $\mathbb{X}(\beta_i) = \mathbf{A}_i$ varies uniformly across the space of polynomials passing through the t' points known by $\mathcal{A}$ for $i \in [\frac{t+t'}{2} + 1 - t']$. The same argument allows us to claim that $\mathbf{B}_i$ and $\mathbf{C}_i$ will be random to $\mathcal{A}$ subject to $\mathcal{E}(\mathbb{z}(\beta_i)) = \mathcal{E}(\mathbb{x}(\beta_i)\mathbb{y}(\beta_i))$. The security property of the encryption scheme allows us to claim that $(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i)$ are unknown to $\mathcal{A}$.

□

C.1.3 Proof of lemma 6 To end this section, we now prove lemma 6 using the subprotocols introduced in §C.1.2 and §B.4.

Proof. Termination: The sharing instances will terminate following the assumption of an initial synchronous round of broadcast. The termination of $TripExt$ ensure that all honest players will terminate the protocol $PreProc$

Correctness This property follows from the correctness property of the $PoPM$ (see below) and $TripExt$.

Privacy: Given that there will be at least $t + 1$ honest players in set $\mathcal{U}$ and that the multiplication triples broadcast by the honest players are random and unknown to $\mathcal{A}$, the privacy property of $TripExt$ ensures that the output triple in $PreProc$ is random and unknown to $\mathcal{A}$.

□

Proof of plaintext multiplication (PoPM)

$$\begin{aligned} R_{PoPM} = \big\{ & c_a \in \boldsymbol{\mathcal{C}} \ , \ c_b \in \boldsymbol{\mathcal{C}} \ , \ c_d \in \boldsymbol{\mathcal{C}} \ ; \\ & A(X) := \sum_i a_i X^i \in \mathbb{F}_p[X]_{\leq t} \ , \\ & B(X) := \sum_i b_i X^i \in \mathbb{F}_p[X]_{\leq t} \ , \\ & D(X) := \sum_i d_i X^i \in \mathbb{F}_p[X]_{\leq t} \ : \\ & R_{PoPK}(a) \ \wedge \ R_{PoPK}(b) \ \wedge \ R_{PoPK}(d) \ \wedge \ a.b = d \big\} \end{aligned}$$

[6]We recall that $Z^{(j)} = \mathcal{E}(f^{(j)}g^{(j)}) \boxplus (-f^{(j)} \boxdot B^{(j)}) \boxplus (-g^{(j)} \boxdot A^{(j)}) \boxplus C^{(j)}$, where $F^{(j)} = X^{(j)} \boxplus A^{(j)}$ and $G^{(j)} = Y^{(j)} \boxplus B^{(j)}$

C.2 On-the-fly Encrypted Random Value Generation

We propose a linear threshold construction to produce an encrypted random value without setup that we introduce in §C.2.1. We then detail in §C.2.3, an implementation in our setting. Finally, we show in §C.2.5 that this construction makes possible the generation of pairs of public/private keys.

Let first define $F_{kg} : Sk \to K$ that goes from a private key space Sk to a a public key space K, as a generic function that derives a public key in K from a private key in Sk. Depending on the type of keys, different circuits can be computed in F_{kg}. For instance, we assume a black-box access to a Pseudorandom function (PRF) with private key space Sk_{PRF}.

C.2.1 Specification of Encrypted Randomness Generator We define an algorithm, denoted TAE.**Rand**, which has a specification close to a Threshold Coin, as introduced in [CKS05, §4.3.]. Each TAE.**Rand** is parametrized by a public coin number, and takes as public inputs a vector pk of public keys. It outputs a TAE.ciphertext c_r of a value $r \in \mathbb{F}_p$, that enjoys the following properties

1. *Robustness*: two distinct calls to TAE.**Rand** with the same coin number, output a TAE.ciphertext of the same r.
2. *Unpredictability* : consider that the Adversary $\mathcal{A}$, which maliciously controls t players, can ask a polynomial number of executions of TAE.**Rand** on coin numbers C_i of his choice, and asks to TAE.**PubDec** for any of the outputs previously produced by these executions. Then, upon choosing a coin number C_i of its choice which was not previously publicly decrypted, $\mathcal{A}$ has a negligible advantage in distinguishing whether it is given a value r' sampled at random in $\mathbb{F}_p$, or, the actual TAE.**PubDec** output r of TAE.**Rand** executed on the coin number C_i.

Notice in particular that robustness implies that, two different Kings executing TAE.**Rand** on the same coin number, output a ciphertext of the same value r.

C.2.2 First implementation following [BHN10], using broadcast TAE.**Rand** can be easily implemented provided an initial synchronous broadcast. The TAE.**Rand**.Contrib then simply consists in every player sample a random plaintext in $\mathbb{F}_p$ then verifiably broadcast a ciphertext of it. The sum of the ciphertexts will be common and random to every player. Our goal is to go beyond this naive idea and to propose a randomness generator that works with asynchronous communications.

C.2.3 Implementing TAE.Rand We are now describing a broadcast-free implementation of TAE.**Rand** that leverages Pseudorandom Secret Sharing (PRSS) that we recall below.

Reminder of Pseudorandom Secret Sharing (PRSS) PRSS enables each players to produce the Shamir share of a random value. The public parameters of a PRSS over $\mathbb{F}_p$, are public sets denoted $s\mathcal{K}_{PRSS}$: the space of secret keys, and $\mathcal{S}$ the space of seeds, a pseudorandom function (PRF) $\psi : s\mathcal{K} \times \mathcal{S} \to \mathbb{F}_p$. The initialization of a PRSS assumes that a trusted dealer gives, to each player, several secret keys as follows. For each subset $A \subset \{1, \ldots, n\}$ of cardinality $n - t$, sample $r_A \in s\mathcal{K}_{PRSS}$ at random, and give it to exactly the players in A. Now, when they need to generate shares of a new random value, then players deterministically select a new seed $a \in \mathcal{S}$ which was not used before, then each player P_l locally outputs

$$PRSS(l, a) := \sum_{|A|=n-t,\, l \in A} \psi_{r_A}(a) \cdot f_A(l) \tag{7}$$

Where f_A is a fixed public polynomial that we do not specify. Then, $PRSS(a)$ is *linearly* reconstructible from any $t+1$ shares.

Construction $\mathsf{TAE}.\boldsymbol{Rand}$ comes as two consecutive steps. The first one takes no input. The second one outputs a $\mathsf{TAE.ciphertext}$, c_s such that the plaintext $s \in \mathbb{F}_p$ is unpredictable for an adversary corrupting at most t players.

The first step takes as parameters a fresh seed a. To be concrete, notice that, in the implementation sketched above in §C.2.1, then a comes as a set of n fresh distinct seeds $(a_i)_i \in \mathcal{S}$. Its contribution function is as follows: each player outputs $\mathbf{Encrypt}(PRSS(j,a))$, along with a proof of correctness, as specfied in C.2.1. Its combination function simply takes as input a set of contributions issued by any set $\mathcal{L} \subset \{1,\dots,n\}$ of $t+1$ distinct players: $\big\{(c^l,\pi^l), l \in \mathcal{L}\big\}$, such that the proofs of correctness π^l are verified, and outputs the concatenation of them along with the proofs.

The second step takes as input such a set of $t+1$ contributions $\{(c^l,\pi^l),\ l \in \mathcal{L}\}$. Let $\lambda_{l\in\mathcal{L}}$ be the Lagrange linear reconstruction coefficients associated to the subset $\mathcal{L}$. Then, the output of this second stage is the linear combination

$$c_{s(a)} := \Lambda_{(\lambda_l)_{l\in\mathcal{L}}}\big(\{c^l\}_{l\in\mathcal{L}}\big)\ . \tag{8}$$

Proposition 12. *The output of these two consecutive steps has the unpredictability property defined as in the game below.*

A proof of proposition 12 is given in C.2.6.

Efficiency consideration We note that the main limitation of the PRSS [CDI05] is that the size of the keys is in $\binom{n}{t}$; however in most practical applications of threshold cryptography, the number of players n is indeed expected to be small.

C.2.4 PRSS Implementation The initial step is to implement the trusted dealer of PRSS keys, by the distributed key generation protocol of §C.2.5. The calls to $\mathsf{TAE}.\mathbf{Rand}$ required in this initial step, can either be implemented with the broadcast, or, recursively, from previous calls to $\mathsf{TAE}.\mathbf{Rand}$ with the broadcast-free implementation that we are describing.

In this implementation, the new space of seeds is $\mathcal{S}^n$. Consider a player l, with inputs its set of secret keys $(r_A)_{l\in A}$, a seed a and $\mathsf{pk}_1,\dots,\mathsf{pk}_n$ the set of public keys. P_l computes $b^l_{i,j} := PRSS(l,a_i)$ on n fixed public distinct seeds: $a_i \in \mathcal{S}$, they are the n coefficients of a polynomial $B^l(X) \in \mathbb{F}_p[X]_t$. Second, it computes the the array of its evaluations, on the α_i for $i \in [n]$, then encrypts the jth entry with j's public key. Third, it produces a proof $\pi_{\mathbf{Rand},j}$ of correct computation of the whole. Namely, of simultaneously: correct evaluation of the $PRSS(l,a_i)$, evaluation at the (α_i), followed by correct encryption.

C.2.5 On-the-fly encrypted randomness generator without broadcast We now enrich PRSS, simultaneously: *encryption of the output* and *public verifiability*, as follows. First, we enrich the secret keys with public keys, namely, we consider: an algorithm $F_{kg} : \emptyset \to (s\mathcal{K}_{PRSS}, p\mathcal{K}_{PRSS})$. Second, we consider a TAE, with plaintext space $\mathbb{F}_p$ and ciphertext space denoted as $\boldsymbol{\mathcal{C}}$, and consider any fixed set of n public keys $\mathsf{pk}_1,\dots,\mathsf{pk}_n$. In what follows, the TAE encryption will be implicitely performed relatively to this set of public keys. We enrich PRSS with a proof algorithm that, on input the set of secret keys $(r_A)_{l\in A}$ of some player l and some seed $a \in \mathcal{S}$, issues a proof that the

(encrypted) output of $\mathbf{Encrypt}(PRSS(l,a))$ is correctly computed. This proof is checked against the set of public keys of player l: $(\mathsf{pk}_A)_{l\in A}$. It is validly checked as soon as all key pairs (r_A, pk_A) are correctly generated with F_{kg}. For sake of concreteness,we illustrate in C.2.4 an implementation of the previous ingredients, based on the one of our TAE in §3.3, and we detail an implementation of the stage in §C.2.3.

Distributed Key Generation We define $\mathbf{KeyGen}_{j,F_{kg}}$ as a set of stages. Informally, it produces a ciphertext $E_j(\mathsf{sk}'_j)$ of a private key $\mathsf{sk}'_j \in s\mathcal{K}$ and the public key $\mathsf{pk}'_j \in K$ derived from sk'_j. This simple idea needs to be carried out on the p-adic decomposition of the sk'_j, since the output of TAE.**Rand** belongs to $\mathbb{F}_p$, and not to Sk. We denote $\log_p |s\mathcal{K}|$ the number of elements of $\mathbb{F}_p$ necessary to encode an element of $s\mathcal{K}$. We define $\mathbf{KeyGen}_{j,F_{kg}}$ as the four followings steps:

1. $c_{\mathsf{sk}_j} \leftarrow$ TAE.**Rand**.*value*: use TAE.**Rand** to produces a vector of TAE.ciphertext denoted as $(c_{sk_j^l})_{l\in 1,\ldots,\log_p |Sk|}$
2. Invocation of TAE.$\mathbf{PrivDec}_j$ on the $(c_{\mathsf{sk}_j^l})_{l\in 1,\ldots,\log_p |s\mathcal{K}|}$. From the output, P_j can deduce his private key sk'_j
3. Evaluation of the circuit which implements F_{kg} applied on the vector $(c_{\mathsf{sk}j,l})_{l\in 1,\ldots,\log_p |s\mathcal{K}|}$ to produce $(c_{\mathsf{pk}_j^l})_{l\in 1,\ldots,\log_p |s\mathcal{K}|}$.
4. Invocation of TAE.**PubDec** to open them, and obtain pk_j by p-adic summation

C.2.6 Proof of proposition 12 The challenging oracle initializes n public/secret key pairs, and samples $\binom{n}{t}$ PRSS keys r_A at random. On each corruption request for an index $j \in [n]$, for a total of at most t indices, the oracle reveals to the adversary the secret key and the $(r_A)_{A\ni j}$. Upon request of a seed a, the oracle returns the $n-t$ correctly computed contributions of uncorrupt keys, then, returns the output $c_{s(a)}$ of the linear reconstruction of the ciphertext coin, as in (8). The guessing advantage of the adversary is the difference between the probability of guessing the value of the plaintext coin $s(a)$, and $1/p$.

Correctness Let us briefly justify that the output of the two stages is indeed a TAE.ciphertext of the shared coin produced by the PRSS on seed a. This is because that (8) applies linear reconstruction homomorphically on TAE-encrypted Shamir shares, and therefore, produces a TAE.ciphertext of the (linear) reconstruction of the Shamir-shared PRSS coin.

Unpredictability Suppose by contradiction that there exists an adversary $\mathcal{A}$ who has nonnegligible advantage in the following predictability game. We are going to show how such a $\mathcal{A}$ can be used to construct an adversary $\mathcal{A}'$ who has nonnegligible advantage against the challenging IND-CPA oracle $\mathcal{O}'$ of TAE, which is a contradiction. $\mathcal{A}'$ initiates the adversary $\mathcal{A}$, and samples $\binom{n}{t}$ PRSS keys r_A at random. From now on, $\mathcal{A}'$ plays the role of the challenging unpredictability oracle towards $\mathcal{A}$. $\mathcal{A}'$ forwards to $\mathcal{A}$ the public keys initialized by $\mathcal{O}'$. On every corruption request for an index j from $\mathcal{A}$, $\mathcal{A}'$ forwards it to $\mathcal{O}'$. Then on response of $\mathcal{O}'$ the secret key sk_j, $\mathcal{A}'$ forwards it to $\mathcal{A}$, along with the RO_j. We assume for simplicity that $\mathcal{A}$ makes exactly t distinct corruption requests, and denote $\mathcal{J} \subset [n]$ their indices. After the corruption phase, $\mathcal{A}$ gives to $\mathcal{A}'$ a challenge seed a. Using the PRSS keys r_A of the $t+1$ uncorrupt players, $\mathcal{A}'$ computes their PRSS shares $PRSS(j,a)_{j\in[n]\setminus\mathcal{J}}$ and deduces the plaintext value $s(a)$. $\mathcal{A}'$ then gives to $\mathcal{O}'$ two challenge plaintexts: $m_0 := a$, and any $m_1 \in \mathbb{F}_p$ distinct from m_0.

Then $\mathcal{O}'$ returns one challenge ciphertext c_b to $\mathcal{A}'$. Now, let us recall that, since $s(a)$ and the t corrupt PRSS shares $PRSS(j,a)_{j\in\mathcal{J}}$ are $t+1$ evaluations of the degree $t+1$ polynomial of the PRSS Shamir sharing, then the uncorrupt PRSS shares are linear combination of them. Let us denote as "Lagrange" the coefficients involved. $\mathcal{A}'$ computes TAE.ciphertext of the t corrupt PRSS shares: **Encrypt**$(PRSS(j,a))_{j\in\mathcal{J}}$, and queries a linear combination on c_b and these t ciphertexts, with the Lagrange coefficients, to deduce $t+1$ prospective uncorrupt encryptions of PRSS shares: $\widetilde{PRSS(j,a)}_{j\in[n]\setminus\mathcal{J}}$, which he forwards to $\mathcal{A}$ as the challenge. Recall that, by construction, if c_b is a TAE.ciphertext of $s(a)$, then these prospective uncorrupt encryptions are exactly TAE.**Rand** contributions of uncorrupt players indices. Therefore, if we are in this case, then $\mathcal{A}$ has nonnegligible distinguishing advantage.

Finally, on output a value m from $\mathcal{A}$: if $m = s(a)$, then $\mathcal{A}'$ outputs $b := 0$ to $\mathcal{O}'$, and otherwise he outputs $b := 1$ to $\mathcal{O}'$.

D New computation structure for proactive security

D.1 Phases description

D.1.1 Contribution phase This phase contains two distinct aspects. On one side each player P_i evaluates a function $contrib_{sid}$ at stage SID, produces *partial proof* π_i and sends a contribution message (noted CONTRIBMSG) to the king. On the other side, upon receiving $t+1$ valid contributions messages associated to a unique SID, the king processes them with the function $combine_{sid}$ in order to compute a ***Combine Proof*** [7] and multicasts the result in a COMBMSG message. Recall that any player can verify a proof using the function $\Pi_{sid}.Verify$.

D.1.2 Verification phase Upon receiving a COMBMSG Z from a king, each player verifies it using a $verif()$ function and, if successful, signs the value contained in the message and sends the result in a VERIFCONTRIB message. This marks the transitions from one stage SID to SID'. When the king received $t+1$ VERIFCONTRIB messages on the same $Z.value$, he concatenates them into a ***Quorum Verification Certificate*** [7]. Then, it appends it to the output of the stage, which is $Z.value$, to form a ***verified stage outputs***, which he multicasts to the players. The function realized by the king that produces a ***VerifOut*** is denoted $verifOutput$. We recall that a player P_i can use its private key to sign a message m, as $\sigma_i \leftarrow sign_i(m)$. Any player can verify any signature using the public keys and the function $SigVerify$.

D.2 Data Structures

Messages. A message m in the protocol has a fixed set of fields that are populated using the $MSG()$ utility shown in algorithm 10. Each message m is automatically stamped with $kingNb$, the king number that leads the computation. Each message has a type $m.type \in \{\text{CONTRIBMSG}, \text{COMBMSG}, \text{VERIFCONTRIB}, \text{VERIFIED-OUTPUT}\}$. $m.sid$ contains the *Stage Identification number* that contains information about the circuit to compute. Finally, $m.value$ contains the material used throughout the computation. There are two optional fields $m.sig$ and $m.proof$. The king uses them to carry respectively the QVC and the CP for the different stages while the slaves used them to carry a partial signature and a ZK proof. We recall that the function to be computed in a stage is embedded in $sid.function$. In summary, players can send four types of messages:

[7] See D for more details

– VERIFIED-OUTPUT: message sent by a king that contains a ***VerifOut*** build from $verifOutput$.
– CONTRIBMSG: message sent by a slave that contains its partial contribution from $contrib_{sid}$.
– COMBMSG: message sent by a king that contains the concatenated contributions and a ***CP*** from $combine_{sid}$
– VERIFCONTRIB: message send by a slave that contains a partial signature of concatenated contributions from $sign$.

Combine Proof. A Combine Proof for a stage SID is a data type that contains the concatenation of individual ZK proofs of correct slave's contributions. Given a Combine Proof cp, we use $cp.kingNb$, $cp.sid$, $cp.value$, $cp.proof$ to refer respectively to the king number, to the stage in which the computation was carried out, to the concatenated result of this computation, and finally to the concatenated proof of correct computation. We note $sid.concat$ the concatenation function. This proof ensures the correctness of the computation.
Quorum Verification Certificates. A Quorum Verification Certificate (QVC) over a tuple $\langle kingNb,\ SID, value, cp\rangle$ is a data type that concatenates a collection of signatures for the same tuple signed by $t+1$ slaves. Given a QVC qvc, we use $qvc.kingNb$, $qvc.sid$, $qvc.value$, $qvc.cp$ to refer to the matching fields of the original tuple. A tuple associated with a valid QVC is said to be a ***verified stage output***.

D.3 Computation structure figure

We show in figure 9 how a stage is carried out for a party P_j. Specifically, we highlight the two phases: first the contribution and then the verification.

D.4 Pseudocode of the structure of computation

The protocols are given in Algorithms 12 and 13. Every party performs a set of instruction based on its role, described as a succession of "**as**" blocks. Note that a party can have more than one role simultaneously and, therefore, the execution of **as** blocks can be proceeded concurrently across roles. Algorithm 10 gives utilities functions used by all players to execute the protocol and algorithm 11 describes specific functions used by the king.

Lemma 8. *(Verification Phase) For Every possible $\mathcal{A}$ and for every possible scheduler, the Verification Phase achieves:* ***(1)*** Termination: *All honest players will eventually terminate.* ***(2)*** Correctness: *For an honest king, the phase outputs a Quorum Verification Certificate.*

Proof. Termination: The honest player P_is will terminate the protocol trivially after sending their contributions to the king. We now argue that an honest king will terminate the protocol as well. Let $\mathcal{A}$ corrupts C players, where $C \leq t$, and let further assume C_1 corrupted players send wrong contributions, C_2 corrupted players send nothing ever and C_3 corrupted players send valid contributions, subject to $C_1 + C_2 + C_3 = C$. Since C_2 players never send any value, the king will receive $t + 1 + C_1 + C_3$ distinct contributions, of which C_1 are incorrect. Since $t + 1 + C_1 + C_3 \geq t + 1$, the king will terminate.
Correctness : This property directly follows the termination property. We have shown above that an honest king is guaranteed to receive at least $t+1$ correct contributions. Thus it is assured to produce a *Quorum Verification Certificate* and to send it to all players. Eventually, all honest players will receive a *Quorum Verification Certificate*. □

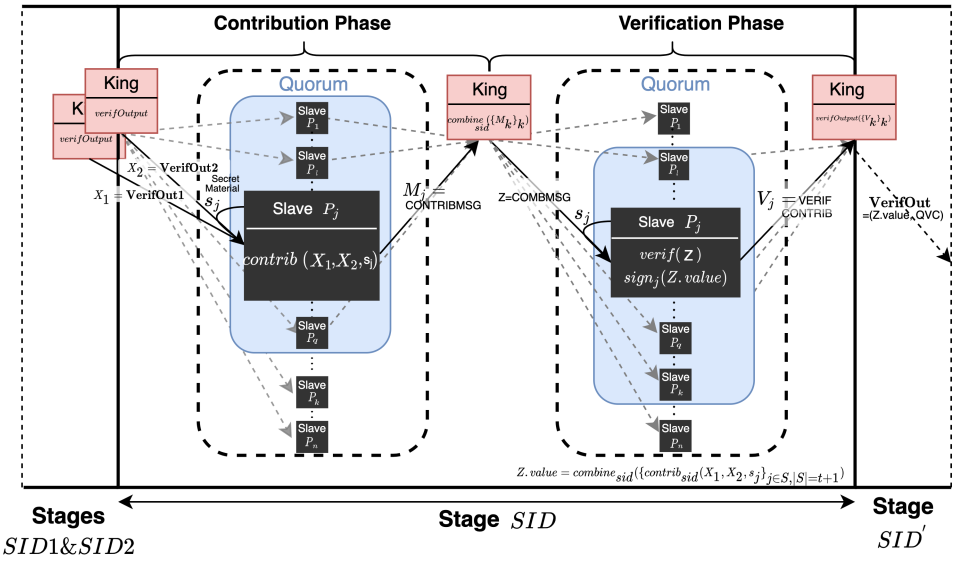

Fig. 9: Computation stage for a party P_j. It first receives two ***verified stage outputs*** X_1 and X_2 from stages SID_1 and SID_2 and uses its secret material s_j to compute its partial contribution using $contrib_{sid}$. The king collects $t+1$ CONTRIBMSG messages with valid proofs, combines the contributions, and sends everything in a COMBMSG message. Finally P_j verifies the proofs and signs the combined contributions and the king concatenates $t+1$ signatures to form a valid output message.

Lemma 9. *(Contribution Phase) For Every possible $\mathcal{A}$ and for every possible scheduler, the Contribution Phase achieves* ***(1)*** Termination: *All honest players will eventually terminate.* ***(2)*** Correctness *phase outputs a Combine Proof*

Proof. The proofs for the *Contribution Phase* are similar to the proofs used for the *Verification Phase* □

E Proactive Security

E.1 Comparison of proactive model with related works

E.1.1 Similarities with [BELO14] The model of [BELO14], is defined under a synchrony assumption where the time is divided into rounds of synchronous communications. The similarity of our corruption model with theirs, is that they also consider separately the specific time periods in which players refresh their shared secrets. They denote these time periods as "refreshment phases", divided between two parts denoted as "opening" and "closing". While in our model above, we denote them simply as "refresh window". The common point with [BELO14], is that a player corrupted while performing a refresh of some epoch e, counts as both corrupt in epoch e and in epoch $e+1$. Anticipating, the rationale for this is that such a player has simultaneously in memory: his plaintexts in clear of all ciphertexts relative to epoch e, and also has his secret decryption key relatively to epoch $e+1$.

Utilities

Function 1 $MSG(type, sid, party, value, sig, proof)$

1. $m.type \leftarrow type$
2. $m.sid \leftarrow sid$
3. $m.value \leftarrow value$
4. $m.sig \leftarrow sig$
5. $m.proof \leftarrow proof$
6. **return** m

Function 2 $verify(m)$

1. if $m.type ==$ "VERIFCONTRIB" or $m.type ==$ "VERIFIED-OUTPUT" :
2. **return** $SigVerify(m.sig)$
3. if $m.type ==$ "CONTRIBMSG" or $m.type ==$ "COMBMSG":
4. **return** $\Pi_{sid}.Verify(m.value, m.proof)$

Function 3 $contrib_{sid}(\{m_i\}_{i \in sid.prev}, secretMaterial)$

1. $V = \{\}$
2. for i in $sid.prev$:
3. if $verify(m_i)$ is True:
4. $V.insert(m_i)$
5. $m \leftarrow MSG(\text{CONTRIBMSG}, sid, \perp, \{m_i.sig\}_{m_i \in V}, \perp)$
6. $m.value \leftarrow m.sid.function(V, secretMaterial)$
7. $m.proof \leftarrow \Pi_{sid}.Prove(V, secretMaterial)$
8. **return** m

Function 4 $sign(value, s_j)$

1. $m \leftarrow MSG(\text{VERIFCONTRIB}, m.sid.number, m.value, \perp, \perp)$
2. $m.sig \leftarrow sign_j(m.kingNb, m.type, m.sid.number, , m.value, , m.proof)$
3. **return** m

Fig. 10: Utilities

king utilities

Function 5 $verifOutput(V)$

1. $qvc.sid \leftarrow m.sid.next : m \in V$
2. $qvc.value \leftarrow m.value : m \in V$
3. $qvc.sig \leftarrow \{m.sig \mid m \in V\})$
4. **return** $qvc.value, qvc$

Function 6 $combine_{sid}(V)$

1. $cp.sid \leftarrow m.sid : m \in V$
2. $(cp.value, cp.proof) \leftarrow m.sid.concat(\{(m.value, m.proof) \mid m \in V\})$
3. **return** cp

Fig. 11: King Utilities

Verification Phase

1. **as** a king:
2. $V = \{\}$
3. Upon receiving a VERIFCONTRIB message m:
4. if $verify(m)$ is True:
5. $V.insert(m)$
6. Wait for $t + 1$ successful verification:
7. $out, qvc \leftarrow verifOutput(V)$
8. Multicasts MSG(VERIFIED-OUTPUT, $m.sid, out, qvc.sig, \perp$)
9. **as** a slave:
10. Upon receiving a COMBMSG message m:
11. if $verify(m)$ is True:
12. Send to king $sign(m, s_j)$

Fig. 12: Verification Phase

Contribution Phase

1. **as** a king:
2. $V = \{\}$
3. Upon receiving a CONTRIBMSG message m:
4. if $verify(m)$ is True:
5. $V.insert(m)$
6. Wait for $t + 1$ successful verification:
7. $cp \leftarrow combine_{sid}(V)$
8. Multicast $MSG(\text{COMBMSG}, sid, cp.value, m.qvc, cp.proof)$
9. **as** a slave:
10. Send to king $contrib_{sid}(\{m\}, s_j)$

Fig. 13: Contribution Phase

E.1.2 Differences with [SLL10]

The first difference is that [SLL10] assumes that players have access to a public-key encryption scheme E which is forward secure. Recall that a forward secure scheme provides local algorithms to update both the public and private keys. However, [SLL10] do not specify how a freshly decorrupted player, who lost all his memory including his decryption key, proceeds to inform all other players of a new public key. Hence, solving this issue would probably require assuming anyway, like we did in §4.5.1, that freshly decorrupted players have access to a public bulletin board of keys at the beginning of each epoch.

This allows us not to make the forward-security assumption. The advantage of not making this assumption, is that we have access to the encryption schemes of Paillier and ElGamal-in-the-exponent, which are both semi homomorphic (§2.4.2), as required by our implementation of 3.3, and also suitable for efficient ZK proofs (§B.1).

The second difference is that in [SLL10], the closing operation of an epoch is not guaranteed to take a predetermined finite number of consecutive exchanges. Indeed, the closing of an epoch succeeds only if a designated player, which they denote "primary", is honest, and benefits from a fast enough network (also known as "partial synchrony" condition). Indeed, they explain in (6) of §5 that, if this primary is not able to have players refresh their shares of secrets in a timely delay, then "the group will carry out a view change, elect a new primary, and rerun the [refresh] protocol." Beyond the issue of performance, this extra delay also brings a safety issue. Indeed, recall that, while they are performing a refresh, players hold both secret shares of the previous epoch and sensitive information about the new epoch. Therefore, a player corrupted during a refresh counts in the corruption budget of both epochs By contrast, our specification the "refresh", which includes the implementation §4.5.2, enables players to erase their keys immediately after the incompressible minimal delay of $\delta_{\mathsf{Refresh}}$.

E.1.3 Differences with Cachin-Kursawe-Lysyanskaya-Strobl [CKLS02] The first difference is that they assume that encryption and decryption are performed locally at each player by a trusted hardware. They furthermore assume that each pair of players creates a new session key

at each epoch, but that the public keys remain unchanged[8]. So this is orthogonal with our computation model without private channels, in which all threshold ciphertexts sent on the network are seen by the adversary. There is also a practical reason for which such a hardware assumption is incompatible with our protocol. It is that our MPC protocol requires players to produce ZK proofs of correct computation on TAE ciphertexts, including decryptions. Players would not be able to produce such ZK proofs if the secret witness, which is their secret decryption key, was concealed in a hardware.

The second limitation is that they assume that all messages sent to a player relatively to epoch e, are delivered to this player while it is in epoch e (page 18 : "Note that this definition guarantees that the servers complete the refresh only when the adversary delivers messages within [epochs]"). Without this constraint, they stress that secrets may be lost during the refresh ("Otherwise, the model allows the adversary to cause the secret to be lost, in order to preserve privacy."). To be sure, we do make this assumption in our simplified model §4.5.1. But then we remove it in §4.5.3

E.2 Details about $\mathsf{ReShare}_k$

The following 2-steps threshold construction, denoted $\mathsf{ReShare}_k$ takes as public inputs: a vector of public keys $\mathbf{pk}$ (the ones of the finishing epoch) and $\mathbf{pk}'$ (the ones of the next epoch), and a $\mathsf{TAE.ciphertext}$ c under $\mathbf{pk}$. Then, out of any $t+1$ correct outputs of $\mathsf{ReShare}_k.\mathsf{contrib}$, there is a public algorithm $\mathsf{ReShare}_k.\mathsf{Combine}$, e.g. run by the king, which outputs a c' under $\mathbf{pk}$ of the same plaintext, such that the adversary's knowledge from the finishing epoch is rendered useless.

Then, correctness and unicity of c' is achieved by using the new computation structure §4.4. Let us recall how the *verification phase* operates on this example, for simplicity we will then omit it from the remaining description. The king forwards c' along with $t+1$ signed outputs of $\mathsf{ReShare}_k.\mathsf{contrib}$ from which it is formed, appended with NIZK proofs of correct computation provided by the contributors. Upon collecting $t+1$ signatures on c', the king appends these signatures to c', which attest unicity and correctness of c'. Such a signed ciphertext is what we denoted as a $\mathsf{TAE.ciphertext}$ ***verified*** $\mathsf{TAE.ciphertext}$.

We recall that a well formed ciphertext comes as a vector of size n, whose entries decrypt to evaluations $B(i)$ of a polynomial $B \in \mathbb{F}_p[X]_t$, the plaintext being defined as $m := B(0)$. $\mathsf{ReShare}_k.\mathsf{contrib}$ is defined as:

$\mathsf{ReShare}_k.\mathsf{contrib}$ **for player** P_i**:** Decrypt the i-th entry of c into m_i, then output $\mathsf{TAE}.\mathbf{Encrypt}(m_i, \mathbf{pk}')$. Concretely, with our implementation: sample a random polynomial $B'_i \in \mathbb{F}_p[Y]_t$ evaluating to m_i at 0, and output the n sized vector of encryptions $\big[c_i^{(j)} := \mathbf{Encrypt}(B'_i(j), \mathsf{pk}'_j,\ j \in [n]\big]$. We do not detail the NIZK of correctness, which can be seen as a combination of the proofs of plaintext knowledge and the proof of correct re-encryption detailed in §B.1.1 and §B.1.3.

$\mathsf{ReShare}_k.\mathsf{Combine}$**:** On input a set of any $(t+1)$ contributions from a $(t+1)$-subset $\mathcal{I} \subset [n]$ of players: $\big[c_i^{(j)},\ j \in [n]\big]$ for $i \in \mathcal{I}$, let λ_i be the Lagrange reconstruction coefficients associated to $\mathcal{I}$. Then, output the vector c' equal to the additively homomorphic linear combination

$$c' := \boxplus_{i\in\mathcal{I}}\big(\lambda_j \boxdot \big[c_i^{(j)},\ j \in [n]\big]\big)\ . \tag{9}$$

[8] "The communication link between every pair of servers is encrypted and authenticated using a phase session key that is stored in secure hardware. A fresh session key is established in the co-processor as soon as both enter a new phase, with authentication based on data stored in secure hardware (if a public-key infrastructure is used, this may be a single root certificate). Thus, even if the adversary corrupts a server, she gains access to the phase session key only through calls to the co-processor."

Proposition 13 (Correctness)**.** *The output of* $\mathsf{ReShare}_k.\mathsf{Combine}$ *from any set of correct* $t+1$ $\mathsf{ReShare}_k.\mathsf{contrib}$ *applied on the same* well formed ciphertext c *of* m*, is a* well formed ciphertext*, under* $\boldsymbol{pk}'$*, of the same plaintext* m*.*

Proof. The proof is a straightforward adaptation of the correctness of [CKLS02], as recalled in §4.5.2. The plaintexts of c' are evaluations at j of the polynomial $\sum_{i\in\mathcal{I}} \lambda_i B'_i(Y)$, whose value at zero is equal to $\sum_{i\in\mathcal{I}} \lambda_i m_i = m$. □

E.3 Privacy: proof of proposition 9 for $\mathsf{TAE.ReShare}_k$

We prove the Proposition in the strongest sense, i.e., we give more power to the adversary in that we provide it with a complete view of all secret informations held by players in epoch e before the Refresh. We are going to show that the view of the adversary of Refresh, is nevertheless computationally independent from its complete view of epoch e. Recall that the total number of corrupt players during both the Refresh and the next epoch $e+1$ cannot exceed t. For simplicity, in the game and in the proof, we consider only one corruption pattern. Namely: the extreme case where the adversary waits to see the new keys pk'_i published, and all the ciphertexts produced by ReShare, before choosing an arbitrary set $\mathcal{I}'_{\mathcal{A}}$ of t players to corrupt in the next epoch $e+1$. As we are going to see in the proof, this "adaptive" pattern incurs an exponential loss in the reduction argument. All other corruption patterns for epoch $e+1$ are easier to handle. [For example, at the other extreme, we have the adversary corrupting in epoch $e+1$ as soon as the clock ticks Refresh. Notice that, in this case, if t players were already corrupt in epoch e, then the adversary must corrupt the same t ones, since we recall that corrupt players during a Refresh count in both corruption budgets. Notice that, in this case, the adversary can choose which new keys the corrupt players publish, but before seing the challenge ciphertexts, so there is no exponential loss in the reduction in this case.] Also, for simplicity, we consider only one instance of $\mathsf{TAE.ReShare}_k$. The general case of n instances in parallel (one for each king) adds no difficulty, since honest players have independent behaviors in each instance.

We formalize the proposition, for simplicity in the aforementionned specific corruption pattern, as the following game. In short, $\mathcal{A}$ can request an oracle, denoted O_R with a plaintext m, then O_R provides $\mathcal{A}$ with the full view of an $\mathbf{Encrypt}(m)$ in some epoch e, followed by all the messages sent during a Refresh operation, followed by t corruptions of $\mathcal{A}$ in epoch $e+1$.

1. $\mathcal{A}$ chooses a set of t indices $\mathcal{I}_{\mathcal{A}}$ and a set of strings of same format as public keys: $(\mathsf{pk}_i)_{i\in\mathcal{I}_{\mathcal{A}}}$ which it gives to O_R.
2. $\mathcal{O}_R$ runs $KeyGen()$ to generate the remaining $t+1$ key pairs: $(\mathsf{sk}_i, \mathsf{pk}_i)_{i\in[n]\setminus\mathcal{I}_{\mathcal{A}}}$ and gives them, both private and public keys, to $\mathcal{A}$;
3. $\mathcal{A}$ chooses a message m, and gives to $\mathcal{O}_R$ a correctly generated encryption $\mathbf{Encrypt}(m)$;
4. $\mathcal{O}_R$ runs $\mathsf{KeyGen}()$ to generate n new key pairs $(\mathsf{sk}'_i, \mathsf{pk}'_i)_{i\in[n]}$. It tosses $\beta \in \{0,1\}$, then:
   (a) if $\beta = 1$, then for each $i \in [n]$, $\mathcal{O}_R$ correctly generates a $\mathsf{ReShare}_k.\mathsf{contrib}$ $c'^{(1,i)}$ of c, which we recall is a n-sized vector of ciphertexts;
   (b) if $\beta = 0$, then for each $i \in [n]$, $\mathcal{O}_R$ generates a n-sized vector $c'^{(0,i)}$ of ciphertexts under the $(\mathsf{pk}'_j)_{j\in[n]}$, of uniform independent random plaintexts in $\mathbb{F}_p$.
5. in both cases, $\mathcal{O}_R$ gives to $\mathcal{A}$ all the new public keys $\mathsf{pk}'_{i\in[n]}$ to $\mathcal{A}$, along with the n vectors;
6. $\mathcal{A}$ gives to $\mathcal{O}_R$ a subset of t indices $\mathcal{I}'_{\mathcal{A}}$, which returns to $\mathcal{A}$ the secret keys $(\mathsf{sk}_i)_{i\in\mathcal{I}'_{\mathcal{A}}}$;
7. $\mathcal{A}$ outputs a bit.

The claim which are going to show, which implies Proposition 9 by the previous discussion, is then that $\mathcal{A}$ has a negligible advantage in guessing β.

Proof. We will show that if $\mathcal{A}$ exists for the game presented above, then we can construct an adversary $\mathcal{A}'$ that has a non-negligible advantage in the $(n-t)n$ messages IND-CPA game for E encryption, as presented in §A.1.3 (there, in the case of $(n-t)$ messages). Let us denote again $\mathcal{O}_E$ the oracle of this $(n-t)n$ messages IND-CPA game. For simplicity, we actually consider the variation to this $(n-t)n$ game (which we could also have done in §A.1.3), in which, when $\mathcal{O}_E$ behaves as the dummy oracle, instead of encryptions of 0, it returns encryptions of uniform independent random plaintexts (in $\mathbb{F}_p$).

The overall strategy of $\mathcal{A}'$ consists in initiating a copy of $\mathcal{A}$, then playing the role of the challenging oracle $\mathcal{O}_R$ towards $\mathcal{A}$, in a way that gives $\mathcal{A}'$ a distinguishing advantage in the $(n-t)n$ messages IND-CPA game.

The $(n-t)n$ messages IND-CPA game game starts by having $\mathcal{O}_E$ generate $(n-t)$ key pairs for E, and give the public keys (pk'_i) to $\mathcal{A}'$. Then, $\mathcal{A}'$ behaves towards $\mathcal{A}$ as $\mathcal{O}_R$ until step (3). Denote $(m_i)_{i\in[n]}$ the plaintext shares of m which are encrypted in c. Recall that they are known to $\mathcal{A}$ and $\mathcal{A}'$ (since the latter can decrypt $t+1$ coordinates, then interpolate the remaining ones).

(4) Then, $\mathcal{A}'$ samples t key pairs and shuffles the indices $i \in [n]$: let us denote $(\mathsf{pk}'_i)_{i\in[n]}$ the list of all public keys after shuffling, in which $\mathcal{I}'_{\mathcal{A}} \subset [n]$ denote the t indices of the ones sampled by $\mathcal{A}$, i.e., of which it knows the secret keys, and $\mathcal{I}'_{\mathcal{A}} \subset [n]$ the remaining ones, i.e. which were given by $\mathcal{O}_E$. Then for each $i \in [n]$, $\mathcal{A}'$ correctly computes the plaintexts of the $\mathsf{ReShare.Contrib}$ of i. Namely, it samples a polynomial $B_i \in \mathbb{F}_p[X]_t$ evaluating to m_i at zero, then computes the n-sized vector of evaluations $[B_i(j),\ j \in [n]]$. Then $\mathcal{A}'$ gives to $\mathcal{O}_E$, for each of the previous vectors, the challenge $(n-t)$-sized vectors of the coordinates in $[n]\backslash\mathcal{I}'_{\mathcal{A}}$. Then $\mathcal{O}_E$ returns to $\mathcal{A}'$ n vectors each of size $(n-t)$ (either encryptions of the actual n vectors, or encryptions of n vectors of uniform plaintexts, in both cases the coordinates of the vectors are encrypted under the $(\mathsf{pk}'_j)_{j\in[n]\backslash\mathcal{I}'_{\mathcal{A}}}$).

(5) Then $\mathcal{A}'$ gives to $\mathcal{A}$ all public keys $(\mathsf{pk}'_j)_{j\in[n]}$ and n vectors of ciphertexts, each consisting of: the $(n-t)$ coordinates in $\mathcal{I}'_{\mathcal{A}}$ equal to the previous challenge $(n-t)$-sized vectors returned by $\mathcal{O}_E$, and the coordinates in $\mathcal{I}'_{\mathcal{A}}$ equal to actual encryptions of the $[B_i(j),\ j \in \mathcal{I}'_{\mathcal{A}}]$.

(6) Upon receiving a corruption request of a subset of t indices from $\mathcal{A}$: if the subset is not equal to $\mathcal{I}'_{\mathcal{A}}$, then $\mathcal{A}'$ outputs a bit at random in the $(n-t)n$ IND-CPA game. Else if the subset is equal to $\mathcal{I}'_{\mathcal{A}}$, then $\mathcal{A}'$ gives the secret keys $\mathsf{sk}'_{\mathcal{I}'_{\mathcal{A}}}$ to $\mathcal{A}$.

(7) $\mathcal{A}$ outputs a bit, then $\mathcal{A}'$ outputs the same bit.

We first claim that the first case in (6) (mismatch of the corrupt set with $\mathcal{I}'_{\mathcal{A}}$) happens $\binom{n}{t}$ times in expectation. Indeed, by $n \times n$ messages IND-CPA, we have that, whatever the secret bit $\beta \in \{0,1\}$ tossed by $\mathcal{O}_E$, the views of $\mathcal{A}$ in both cases, at the point where it decides the t-set to corrupt, are computationally undistinguishable.

Then, to conclude the argument we have that, in the second case of (6), then:

- When $\mathcal{O}_E$ tosses $\beta = 1$, then the view of $\mathcal{A}$ is exactly the same one as if facing $\mathcal{O}_R$ in the case where $\mathcal{O}_R$ samples $\beta = 1$;
- When $\mathcal{O}_E$ tosses $\beta = 0$, then the view of $\mathcal{A}$ is: the nt-set of the plaintext coordinates in $\mathcal{I}'_{\mathcal{A}}$ of all n vectors, vary uniformly independently in $\mathbb{F}_p^{nt}$ (by property of Shamir secret sharing, recalled in Property 11, applied to each secret m_i). In addition, the $n(n-t)$-set of the plaintext coordinates in $[n]\backslash\mathcal{I}'_{\mathcal{A}}$ of all n vectors, vary uniformly independently in $\mathbb{F}_p^{n(n-t)}$, independently of the previous nt set. Thus, the view of $\mathcal{A}$ is the same one as if facing $\mathcal{O}_R$ in the case where $\mathcal{O}_R$ samples $\beta = 1$.

Thus by assumption $\mathcal{A}$ has nonnegligible advantage in distinguishing the secret bit β of $\mathcal{O}_E$, and thus also $\mathcal{A}'$ by construction. $\square$