Almost-Asynchronous MPC under Honest Majority, Revisited

Matthieu Rambaud and Antoine Urban
Télécom Paris and Institut polytechnique de Paris
Palaiseau, France
matthieu.rambaud,antoine.urban@telecom-paris.fr

Abstract

Multiparty computation does not tolerate $n/3$ corruptions under a plain asynchronous communication network, whatever the computational assumptions. However, Beerliová-Hirt-Nielsen [BTHN10, 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 [BTHN10] suffers from 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 [BTHN10, Podc'10] 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:* the previous limitation directly precludes the possibility of tolerating a mobile adversary. Indeed, tolerance to this kind of adversary, which is denoted as "proactive" MPC, would require, in the above setup, a mechanism by which players refresh their secret shares of the global key, which requires *on-the-fly* generation of common randomness. (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.

Of independent interest, the novel computation framework that we introduce for proactivity, revolves around players denoted as "kings", which, in contrast to Podc'10, are now *replaceable* after every elementary step of the computation.

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 ([BTH08; Esc+20]). 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 [Gär99; Dam+09; HNP05; CHP13; Bac+14] were developed for the *asynchronous* communication model. This setting comes with limitations: Ben-Or, Kelmer, and Rabin [BOKR94] 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.

In [BTHN10], Beerliová-Trubíniová, Hirt and Nielsen observed that one initial synchronous broadcast round is sufficient to enforce input provision and tolerate $t < n/2$ corruptions in an almost-asynchronous network. More precisely, they show that the minimum assumption is that players start with a consistent view of encrypted inputs. This is relevant in a use-case where players publish encryptions of their inputs on a ledger, to demonstrate their interest in taking part to a MPC computation. Then the actual computation can go offline and asynchronously 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. Players in their protocol broadcast threshold encryptions of their inputs along with proofs of plaintexwt knowledge, 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 [Abr+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 [BTHN10, §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" [Daz+08; 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 only make the two standard assumptions $\mathcal{F}_{PKI}$ and $\mathcal{F}_{ZK}$ detailed in §2.3. The first is a public bulletin board where players can publish their public key ([BCG21; TLP20]) ahead of the execution. The second enables to prove knowledge of a witness satisfying some relation, without revealing more about it. Let us mention that these are the necessary and sufficient functionalities for decentralized confidential transaction systems [Lai+19].

Theorem 1. *(Informal) Assuming $n = 2t+1$ players in an asynchronous communication network, of which t are maliciously corrupted by a polynomial adversary, in the $\mathcal{F}_{PKI}$ and $\mathcal{F}_{ZK}$ hybrid model; Given consistent views of the encrypted inputs ("input predistribution"), then*

any arithmetic circuit over any (small) finite field $\mathbb{F}_p$ can be securely computed over an asynchronous network, with input provision.

Furthermore, augmenting the model with $\mathcal{F}_{NIZK}$, then input predistribution can be trivially implemented in one synchronous broadcast round, of size $O(n^3)$ bits sent per player.

In what follows (§1.1 and §1.2.2) we highlight the technical hurdles with respect to previous works, and give an overview of the proof of Theorem 1. Then in §1.2 we show how we solve all the other limitations presented in (ii) (iii) and (iv) of the abstract.

1.0.1 Roadmap of the Proof of Theorem 1 We first stress in §1.1.1, §1.1.2 that, in our demanding model of transparent setup with asynchrony, then previous transparent threshold encryption schemes support only a finite number of homomorphic additions, due to growth of the plaintexts, and, in §1.1.3, that known techniques for fixing this problem, known as "bootstrap" or "refresh", fail here. We then sketch in §1.1.1 the main novel ingredient that we introduce to solve Theorem 1. Namely: a threshold encryption scheme (TAE) operating in our demanding model, that supports an unlimited number of additively homomorphic operations, at the cost that these operations are now performed by a $(t+1)$-threshold mechanism. In particular, instead of a single global public key generated by a trusted setup, TAE takes as parameters all the n public keys published by the players ahead of the execution (adapting it to the case where up to t keys are not published is straightforward). In §2.1 we detail the model, in §2.2 we recall the baseline protocol of [BTHN10], in 2.3 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; Cho+13]). On the other hand, how to achieve threshold encryption *with a transparent setup* 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 ZK proof of correctness. This is suggested for the first time by Goldreich et al. [GMW91, §3.3], where it appears as wrapped into a scheme to verifiably share a secret in one single round of broadcast. Remarkably, this has been independently re-discovered by three other research streams: first by [Sta96], in which it is formalized as *Publicly Verifiable Secret Sharing* scheme (PVSS), followed by [FO98; Sch99; BT99; YY01] [CS03, §1.1] [RV05; HV08; JVS14]; 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 [Daz+08], followed by [CFY16] [RSY21, Appendix E].

1.1.2 Previous Limitations in the Number of Homomorphic Additions, due to Growth of Size of the Plaintext Since we follow the blueprint of the MPC protocol [BTHN10], we need to support homomorphic additions on the ciphertexts. The straightforward idea to achieve this is to instantiate the previous PVSS, with additive encryption. Unfortunately, all of the previous works which applied this idea, ended up with supporting only a limited number of additions. 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 Paillier encryption: we recall and formalize both in §B.2.2. Let us illustrate the roadblock encountered by these previous works.

A. el Gamal-in-the exponent the PVSS of [Sch99, §5], which is applied to electronic voting, is instantiated with the el Gamal in the exponent scheme. 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 addition, 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.".

B. 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, homomorphic additions of PVSS are guaranteed to decrypt correctly only if the plaintexts' sizes remain smaller than $N/2$, where N is the minimum of the N_i. This is why it is said in [RSY21] that this PVSS *"supports a limited number (currently set to n) of homomorphic additions"*.

C. 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; BTHN10], then all plaintexts belong to a fixed $\mathbb{Z}/N\mathbb{Z}$ with *unique* N. This guarantees unlimited homomorphic additions modulo N in this single ring of plaintexts. This thus enables MPC over $\mathbb{Z}/N\mathbb{Z}$.

D. Semi Homomorphic Encryption (SHE) Bendlin et al.[Ben+11, Section 2] coined the notion of SHE, to denote any public key encryption scheme, not necessarily threshold, that supports a limited number of additions. Namely, a SHE guarantees correct decryption modulo p as long as the size of plaintext is below a bound. Concretely, in Paillier, when many additions bring the size of the plaintext above N then the decryption modulo p is *not* equal to the plaintext modulo p (in fact the bound is $(N-1)/2$, for sign reasons). Such limitation also concerns Regev's LWE based cryptosystem [Reg09] or Gentry, Halevi and Vaikuntanathan's scheme [GHV10] (which have also a bound on the randomnessfor correct decryption modulo p). To this list we add what we denote as ElGamal in the exponent (see §B.2).

1.1.3 Technical Hurdles with 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 mentionned issues, one could think of the following mechanism.

First attempt. At regular intervals, each honest player i would decrypt his 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 his 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).

Second attempt. One could think of the interactive mechanism, proposed by Choudhury-Loftus-Orsini-Patra-Smart, under the name "refresh" in Figure 3 of [Cho+13]. However, their mechanism leaves unchanged the plaintext: it reduces only the noise. Thus, it is orthogonal to our concern. Notice also that, since [Cho+13] assume a trusted setup, their scheme enjoys a single public encryption key, which we cannot in our setting.

1.1.4 Our solution: bivariate PVSS for unlimited threshold additions

Overall idea From the first attempt, we see that the challenge is to make possible that, after an unlimited number of additions, the PVSS share of *every* player remains of small size. To achieve this, instead of a PVSS equal to a vector of shares, as in all previous schemes, we introduce in §3.3 a novel construction of PVSS, that uses for the first time a *double-sharing*. This PVSS allows the construction of a mechanism for unlimited homomorphic additions of ciphertexts. In detail, addition of ciphertexts is now a threshold mechanism, just as threshold decryption. Namely, on input two ciphertexts, any player can output what we denote an "addition share", using an algorithm denoted as **Add**.Contrib, along with a ZK proof of correctness. Then, there is a public algorithm, denoted **Add**.Combine, that, on input correct addition shares of the same two ciphertexts from any $t+1$ distinct players, outputs a ciphertext of the sum.

Thanks to the bivariate structure of the PVSS, the addition shares produced by any $t+1$ players, after *any arbitrarily large number of additions*, contain enough material to enable the t remaining players to reconstruct their share of the sum, such that these shares also have *small* plaintext sizes.

Furthermore, one can actually remove most of the interactions, by instantiating with a semi homomorphic encryption scheme (SHE: see above). We denote the overall scheme as "Threshold Additive Encryption" (TAE), for which we provide the details in Section 3.

Details of the technique In detail, each share of our PVSS, now, comes now as a n-sized column vector of ciphertexts, such that we have the following symmetry. For every player j, then for each index i, then the plaintext of the i-th entry of his column is equal to the plaintext in the j-th entry of the column of player i. Thus, when $t+1$ players add modulo p the plaintexts of their columns of ciphertexts, then by symmetry, they are also able to *fill* the $(t+1)$-corresponding lines. This maintains the invariant that *each* column contains at least $t+1$ entries with plaintexts reduced modulo p, and thus, that *any* player can decrypt $t+1$ plaintexts on his column, which is *enough to recover his whole column, by interpolation.*

Next, one can instantiate the baseline encryption scheme with a SHE (see above, such as Paillier, or el Gamal in the exponent). This, in turn enables noninteractive homomorphic additions on TAE ciphertexts, up to a certain bound on the sizes of plaintexts. When the bound is reached, then players perform again an interactive additions with the previous mechanism. Thus, the plaintext of the output is again brought back to small size $[0,\dots,p-1]$. Thus, further noninteractive additions are again enabled.

1.1.5 Epilogue: another tradeoff, with BGV. Let us make here the simple but apparently new observation that, instantiating PVSS with the encryption scheme of [BGV12], enables unlimited additions, without the need of any interactive mechanism. 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 [BGV12] 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 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 [BTHN10] 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 §4.1. First, we require every player to append a NIZK 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 $(t+t')/2-t'$ unpredictable triples, and thus of *at least one.*

Theorem 2. *(Informal) Assuming an initial round of broadcast, possibly the same as the one of Theorem 1, then players can produce*

common encrypted random multiplication triples unknown to the adversary, in a fixed (constant) number of consecutive interactions.

1.2.2 Computation method We propose a novel computation framework suited for asynchronous MPC with proactive security. We abstract out the structure of computation of [BTHN10], as follows, which defines our baseline. 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 others, and as a slave for other $n-1$ instances computing the same circuit. This model of computation guarantees that all instances relating to an honest king give at the end of the protocol all correct outputs are the result of the same set of instructions. This is further enriched with a new verification structure. In more detail, we define an atomic step of computations, which we denote as "Stage". It maintains the invariant that it outputs a result signed as valid by $t+1$ players and takes as input validly signed outputs of other stages. In other words, checks are chained throughout the process and not pushed at the end of the protocol as formally explained in §4.2.1. This is the main difference with [BTHN10]. We motivate this choice of intermediary checking for two reasons. First, it simplifies the termination process. Unlike in [BTHN10], upon receiving a correct output, a player multicasts it and immediately terminates. Second, it enables proactive security. To this end, it is indeed necessary that any player can take over the role of the king while being certain of the validity of the calculations undertaken so far. More details are provided in section 4.4.

1.2.3 On-the-fly generation of threshold-additive encrypted randomness The generation of a common random encrypted secret was proposed in [BTHN10]. It naively consists of asking each player to generate a random element, broadcast an additive encryption of it in the first round, which are then summed together. We remove the dependency on the broadcast. Our protocol, described in section 4.3 can indeed be executed at any moment in an asynchronous setting. 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 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].

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" and we assume that the adversary can corrupt at most t players in two consecutive epochs. The set of corrupted players may change from one period to the next, so the protocol must remain secure, even though every player may have been corrupt at some point. In the context of our encryption scheme, which is a vector of n ($(t+1)$-sized) encrypted shares, this model adds a triple threat. First, if players do not change their secret keys and reencrypt the shares at regular intervals, then, the adversary may use the keys of newly corrupted players, to decrypt their share of a ciphertext of which he previously gained knowledge of t other shares. To address this first threat, we deduce an *on-the-fly* new keys generation mechanism, without setup, from the encrypted randomness generator introduced above. The second threat is that the model assumes that newly de-corrupted players lost all their memory, in particular their secret keys. To address the latter issue, each player generates a new public / private key pair at the end of each epoch, which leverages either our new key generation mechanism for those who have their memory intact, or the bulletin board. Players need to be able to decrypt their shares of the PVSS in any epoch. Following the seminal work of [Her+95] on proactive security, different protocols, e.g., [ZSR05] [SLL10] [Mar+19] based on resharing have been proposed, but are not directly applicable in our setting as they either require broadcast or Byzantine agreement, i.e. consensus. Finally, the third thread is that, re-encrypting the n shares (each $(t+1)$-sized) constituting a TAE ciphertext, with freshly generated public keys is not enough. Indeed, recall that these plaintext shares are evaluations of a polynomial (bivariate symmetric, in our scheme). Thus, a mobile adversary can decrypt, after 2 epochs, enough evaluations of the polynomial to interpolate the value at $(0,0)$, which is equal by definition to the TAE-encrypted value. In section 4.4 we detail an interactive protocol to deal with these last two threats simultaneously as follows: it first re-randomize the polynomial, and then reencrypt it with the new keys.

Notice that the bivariate structure of our PVSS makes it possible for any $t+1$ players to securely generate the new encrypted share of any player P. By contrast, with a classical univariate PVSS, this would have lead to disclose their own encrypted share under P's public key.

1.3 Related Work since [BTHN10]

Following the initial result of Beerliová-Trubíniová, Hirt and Nielsen [BTHN10] to achieve security for an honest majority in the almost-asynchronous model, different improvements have been proposed. On the one hand, Cohen [Coh16] removes the need for the costly king/slaves communication paradigm, but requires a trusted setup for its threshold encryption, which we want to remove. Also, [Bac+14] assumes trusted hardware to reach honest majority. On the other hand, advances have been made to reduce the setup assumptions of threshold homomorphic encryption schemes. Recently Damgård at al [Dam+21] proved that the threshold encryption scheme of Gordon et al. [DGLS15] satisfies our setting of an almost asynchronous network, with only a PKI and CRS. Noticeably, they build on the malicious-security compiler of [Ash+12], which itself uses the UC NIZK-PoK of [DS+01] that we recall in 2.3.2. However, their "expanded" ciphertexts consist of vectors of dimension the number of players, whose entries are so-called "expanded ciphertexts", of

size quadratic in the number of players and polynomial in the depth of the circuit. Recall that a minimum order of magnitude is that ciphertexts are contained in lattices of dimension 2^{12} (for hardness of LWE). Finally, we observed that the threshold FHE scheme without trusted setup of Badrinarayanan et al [Bad+20] could be used in the almost asynchronous setting, but its ciphertexts are of size polynomial in both the number of players[1] and in the depth of the circuit, which is a significant space overhead.

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 §4.4. 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 the standalone security model. 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 [BTHN10] We assume that all players have access to a synchronous broadcast channel at the starting time of the protocol. Namely, they have the guarantee that when they send a message on this channel at time $t = 0$, then it will be identically delivered to all players at time $t = \Delta$ where Δ is a public fixed parameter. 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 [BTHN10, PODC'10]

Let us review in more details the MPC protocol of [BTHN10] outlined in the introduction. It securely compute any arithmetic circuit over $\mathbb{F}_p$ in the almost asynchronous model, as detailed in §2.1.1, and tolerates $t < n/2$ corruptions. Moreover it enforces *input provision*. Let us denote $\mathcal{E}$ a threshold encryption scheme with plaintext space $\mathbb{Z}N\mathbb{Z}$ with N a large composite integer, as recalled in §1.1.1, that furthermore comes with a public *noninteractive* algorithm, denoted $\boxplus$, that computes the homomorphic addition of any two ciphertexts. The definition of such a scheme, denoted AHE, is recalled in §A. 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 out model are presented in figure 3.

(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. In particular, thanks to the multiplicative triples, the multiplication gates are brought down to threshold decryptions and homomorphic additions, by the technique of [Bea91]. At each interactive step, the slaves prove to the king that the calculation is correct.

(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

We define and discuss in more detail our setup In §2.3.2 we discuss instantiations of ZK-PoK with a transparent setup. In §2.3.3 we compare our setup with the trusted one of [BTHN10].

2.3.1 $\mathcal{F}_{PKI}$ We make use of a public "bulletin board" of public keys ([BCG21; TLP20]) as presented in figure 1. Notice that this is actually very close to the functionality denoted certification authority $\mathcal{F}_{CA}$ in Canetti [Can04]. Each player can give at most one public key to this bulletin board. The bulletin board outputs, when

[1] Each decryption key share embedded in the ciphertext consists of $O(n^{4.2})$ [Gol20] equivalent of a regular key in order to represent a threshold access structure in terms of monotone Boolean formula.

queried, the public key received from any given player. Concretely in our protocol, players are instructed to publish their public keys. At least all honest players ($\geq t+1$) will do so, since the functionality enables them to. We stress that $\mathcal{F}_{PKI}$ does not perform any checks on the registered value; it simply acts as a public bulletin board.

$\mathcal{F}_{PKI}$

Registration On input (register, sid, v) from player P_i that has not previously registered, record the tuple (P_i, v).

Retrieval On input (retrieve, sid, P_i) from player P_j, if there is a previously recorded tuple (P_i, v), then return (sid, P_i, v) to P_j. Otherwise return $(sid, P_i, \perp)$.

Figure 1: PKI functionality

2.3.2 $\mathcal{F}_{ZK}$ The zero-knowledge functionality $\mathcal{F}_{ZK}$, as introduced in [Can+02], is presented in figure 2. As stressed in [Can+02], it 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 (prove, sid, x, w) from P, ignore if $R(x, w) = 0$. Otherwise send (verification, sid, x) to $\mathcal{A}$ and V and halt.

Figure 2: Zero-knowledge functionality

Let us discuss some instantiations with transparent setup, on the example of [DS+01], which enjoys many properties.

First, they consist in a single message from the prover, and thus are noninteractive (NIZK).

Second, they implement $\mathcal{F}_{ZK}$ 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 [Can+02], [Ash+12, p497] and [Coh16, §4.2].

Third, they require only that players are provided with a *public uniform* random string, which *needs not* be generated with any trapdoor. This minimal assumption is also the one required by [AC20; ACR21] (random groups elements being necessary to implement Pedersen commitment). See §B.3 for a concrete illustration of [ACR21], which has logarithmic size in the statement proven, in our setting. This minimal assumption is also the one of [BS+18], which is the proof system deployed in Ethereum. As nailed by the latter (footnote 8), randomness is actually necessary in any ZK proof system. How to implement a public distributed random beacon with transparent setup is well studied [CD20; Das+21].

Fourth, the public uniform string can be safely *re-used* identically in multiple executions. Thus they achieve the stronger primitive which [Can+02] 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. Alternatively, in a model where the random oracle is initialized by players (e.g., drawn from a hash function family using a random beacon), which enables the simulator to reprogram it (in order to simulate Fiat-Shamir). These assumptions are discussed in [Pas03; Can+07]. Although implementing $\mathcal{F}_{ZK}$ is orthogonal to this work, since it is part of our model, for completeness we discuss in §B.8 implementations of it with a global setup.

2.3.3 Comparison with the setup of [BTHN10] The setup in [BTHN10] assumes a trusted entity that generates *secret* keys (shares) and gives them to players. If the adversary learns these secrets, then the security is ruined. By contrast, the $\mathcal{F}_{PKI}$ 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 [BTHN10] also require a trusted setup, by contrast with the ones recalled in §2.3.2. First, [BTHN10, §5.1-5.2] assumes a secret sharing of a secret key by a trusted authority. Then, in [BTHN10, §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. Recall by contrast that the random string mentionned in §2.3.2 is a mere public coin. Namely, it needs only be uniform, so *needs not* be generated with a secret trapdoor.

2.4 Cryptographic primitives

2.4.1 Shamir secret sharing We denote $\mathbb{F}_p[X, Y]_{(t,t)}$ the ring of bivariate polynomials with coefficients in $\mathbb{F}_p$, of degree bounded by t in both X and Y. Let us recall quickly the *secret sharing scheme of Shamir* over $\mathbb{F}_p$. We consider $\alpha_1, \ldots, \alpha_n$ fixed public nonzero distinct values in $\mathbb{F}_p$, denoted as the *evaluation points*. For instance: $[1, \ldots, n]$. 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 $[f(\alpha_1), \ldots, f(\alpha_n)]$, denoted the "shares". It has the property that, for any fixed secret m, then any t shares vary uniformly. 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 reconstruction 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 Public key encryption with common plaintext space $\mathbb{F}_p$ We say that a public key encryption scheme has common plaintext space $\mathbb{F}_p$ if all plaintext spaces contain $\mathbb{F}_p$. In what follows we actually formalize it as if the plaintext space is $\mathbb{F}_p$. We provide the two main examples (Paillier and el Gamal) in §B.2. Precisely, such a scheme consists in the following triple of algorithms (KeyGen, E, Dec). Let $(s\mathcal{K}, p\mathcal{K})$ be spaces, denoted as the secret and public key spaces. To each $\mathsf{pk} \in p\mathcal{K}$ corresponds what is denoted C_{pk} the "ciphertext space", for instance $(\mathbb{Z}/\mathsf{pk}^2\mathbb{Z})^*$ in Paillier. Let $\mathsf{KeyGen} : \emptyset \to (s\mathcal{K}, p\mathcal{K})$ a PPT algorithm.

Let E be an efficiently computable PPT algorithm with source equal to $(p\mathcal{K}, \mathbb{F}_p)$, and target C_{pk}. Fix any pk output by KeyGen. Let Dec be an efficiently computable algorithm, with source the

union of all $(\mathsf{sk}, C_{\mathsf{pk}})$, where $(\mathsf{sk}, \mathsf{pk})$ is an output of KeyGen, and with target $\mathbb{F}_p \cup \{\text{abort}\}$. We require the completeness condition, that $\mathrm{Dec}(\mathsf{sk}, E(\mathsf{pk}, m \in \mathbb{F}_p)) = m$. We require the classical IND-CPA *privacy* property, defined by the negligible advantage of an adversary to guess between two plaintexts in $\mathbb{F}_p$ of his choice, upon being given encryption of one of them.

3 Proof of Theorem 1

3.1 Overview of Threshold-Additive Encryption (TAE) with Transparent Setup

We sketch a *Threshold-Additive Encryption* (TAE) scheme with *transparent setup* (TAE), following the program presented in §1.1.4. We first sketch below a toy model where the plaintext space is $\mathbb{F}_p$, but every additions are interactive. From the description below it is easily seen that a linear combination can actually be done at once. The formalism of the specifications are provided in §3.2, and the details of the implementation in §3.3, but the following paragaphs should be enough for the comprehension. The advantage of this toy model is that it can be instantiated from any public key encryption scheme E (with plaintext space containing $\mathbb{F}_p$, as in §2.4.2) and is also easier to describe. Then in 3.1.2 we sketch how to enable noninteractive additions, when using a baseline encryption which is semi-homomorphic.

3.1.1 Toy model of TAE *with interactive additions* To encrypt a plaintext s, 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 of B as s. Finally, output the c_s consisting in the $n \times n$ array of E-ciphertexts of evaluations of B: $E_{\mathsf{pk}_j}\big(B(i, j)\big)$, where only $t + 1$ rows need to be actually computed. The entries on the other rows are set to the empty string $\perp$. We denote as TAE.ciphertext this data structure. Such $t+1$ filled rows will actually be an *invariant* of all TAE.ciphertexts, including the TAE.ciphertexts output by interactive operations. The interesting point is that the indices of the filled rows may possibly be *different* between two different TAE.ciphertexts. This invariant is formalized in Def 5 and Prop 9. It is that *all* TAE.ciphertexts, have (exactly) $t + 1$ filled rows. They furthermore consist in encryptions of the evaluations in $[0, \ldots, p - 1]$ of a single bivariate polynomial in $\mathbb{F}_p[X, Y]_{(t,t)}$.

Let us describe how any $t + 1$ players can produce, in one round, a homomorphic addition of two TAE.ciphertext c_a and c_b of some $a, b \in \mathbb{F}_p$. Let us denote K the entity who is meant to obtain the resulting TAE.ciphertext c_{a+b} (the "king" in our protocol). Denote B_a and B_b the underlying bivariate polynomials. Players do not know them since evaluations are encrypted. But their goal is to collectively produce a $n \times n$ array of evaluations of $B_{a+b} = B_a + B_b$, such that $t + 1$ rows are nonempty. Each player i decrypts the $t + 1$ nonempty entries on its column i of c_a and of c_b. Notice that the $t + 1$ indices may possibly *not* be the same on c_a and c_b. From these, it interpolates the degree t polynomials $B_a(X, i)$ and $B_b(X, i)$. By evaluation, it is thus able to reconstruct all the t missing entries on both i-th columns. Notice here, that this i-th plaintext column $\big[B_a(i', i)\big]_{i' \in [n]}$ actually constitutes a decryption share of c_a (likewise for c_b). We denote later the previous steps as "**PubDec**.Contrib", whose output is this decryption share. Indeed, from $t+1$ such shares, it is possible to interpolate the whole B_a, and thus recover a. But in our concern of computing a homomorphic addition, players are not meant to learn a nor b. Thus, they continue local computations as follows, without revealing these decryption shares.

Summing coordinate by coordinate mod$p$, these two (now full) i-th columns, it deduces the full column of evaluations of $B_{a+b}(X, i)$. Which is, by symmetry of B_{a+b}, equal to the row of evaluations of $B_{a+b}(i, Y)$. It thus sends to K its *addition share*, which consists of the encrypted i-th row $\big[E_{\mathsf{pk}_j}\big(B_{a+b}(i, j)\big)\big]_{j \in [n]}$. We denote later as "**Add**.Contrib", whose output is this addition share. Upon receiving $t + 1$ addition shares, K puts them in one single $n \times n$ array , leaving the remaining t rows empty. This constitutes an array of evaluations of B_{a+b}, of which $t + 1$ rows are filled, thus by definition a TAE.ciphertext c_{x+y}.

Notice that previously considered PVSS / ad hoc encryption (§1.1.1) operated instead with a univariate polynomial and encryptions consisting in one n-sized vector of encrypted evaluations. In this setting, assuming that c_x and c_y each contain $t + 1$ nonempty entries, the previous method would have failed to return a vector with $t + 1$ entries, since possibly they have only one nonempty entry in common, so then only one player would have been able to compute the sum of its shares.

3.1.2 Making additions noninteractive. The basic idea is that *Instantiating* the previous TAE with a baseline additive encryption scheme E enables anyone having c_a and c_b with same $t + 1$ filled rows to compute their homomorphic sum. On the face of it, this simplifies a lot the previous construction. But many pitfalls remain. The flagship example is Paillier encryption: recall that in our setting players have *different* public keys, so different actual plaintext spaces $\mathbb{Z}/N_i\mathbb{Z}$. Thus, the only consistent way to define a bivariate polynomial is to consider a *common additive subset* of all these plaintext spaces, which in our case is a (small) field $\mathbb{F}_p$, identified with $[0, \ldots, p - 1] \subset [0, \ldots, N_i - 1]$. But then, if too much noninteractive additions are done, then the plaintexts may go large then wraparound modulo one of the N_i. Then, decryption is not guarantees anymore correct modulo p. This limitation on the plaintext size, wrt correct decryption modulo p, is what [Ben+11] denote as "semi homomorphic" encryption (SHE). In the appendix §B.2 we also observe that a variant of el Gamal, when plaintexts are encoded in the exponent, is also a SHE. Indeed, decryption is computable up to a certain plaintext size. To overcome this issue, players have to make sure to regularly bring back down the size of the plaintexts after many noninteractive addition gates. This can be done simply by any $t + 1$ player. Each player i decrypt its column of c_a, interpolate the univariate polynomial $B_a(X, i) = B_a(i, Y)$ modulo p, then compute its evaluations modulo p and re-encrypt them. Said otherwise, perform an interactive addition with a ciphertext of 0. The collective output of this operation is an encrypted array c'_a, whose plaintext entries are now brought back to $[0, \ldots, p - 1]$, and whose underlying bivariate polynomial B'_a is equal to B_a modulo p. Thus, the plaintext a remains the same modulo p, which is what we wanted. We can formalize this as an interactive "Resizing" mechanism. Notice that this Resizing mechanism must be done *at once on all ciphertexts* that may serve later. Indeed, it is important that all Resized TAE.ciphertext output contain the *same* $t + 1$ nonempty rows. Otherwise, noninteractive additions between them would

not be possible. Thus, this is a *stronger invariant* than the previous one, where the $t+1$ nonempty rows could be different from one ciphertext to another.

Notice that [Cho+13] also describe an interactive bootstrapping mechanism, but the comparison stops here. Indeed, what they reduce is the size of the noise, not of the plaintext. Notice also that they assume a trusted setup, which enables a single public threshold encryption key in their scheme.

3.1.3 No Robustness We *do not* require for Robustness in our specification of TAE Namely, we do not require that the validity of contribution can be verified, nor that K cannot exhibit a ciphertext with two distinct sets of decryption shares opening it to two distinct plaintexts. This is a requirement of *verifiable* threshold homomorphic encryption, as used in [CDN01] and recalled in appendix §A. The reason 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 contributions (see also above) The reason for this modularity is that we need different levels of robustness. For the purpose of Theorem 1 only, it will be specified in the MPC protocol that players prove in ZK to the king K that they correctly computed their shares (of decryption or addition). But there, as in the MPC protocol of [BTHN10], it will *not* be needed that the king prove to slaves the correctness of his decryptions, he *needs not even exhibit the shares* from which he claims to have computed the decryption. By contrast, in the proactive MPC protocol (§4), both players and K will incorporate NIZK proofs of correctness in their shares and aggregation. This will enable a quorum of $t+1$ players to *validate* well-formedness of ciphertexts output by a gate by putting their signature. These validations will enable freshly decorrupted players to take over the computation on the fly.

3.2 Specification of a TAE (toy model)

We formalize the specification required for the above toy model of TAE (3.1.1) where all additions are interactive. In §3.3 we detail the implementation sketched above of the toy TAE from *any* public key encryption scheme in the sense of 2.4.2. The reason is that specifying a TAE with noninteractive additions (as in §3.1.2) would require more formalism: as [Ben+11], specifying that plaintexts are in $\mathbb{Z}$, to parametrise the growth of their size after additions, and to specify that correctness of decryption modulo p is guaranteed up to a certain plaintext size; plus, specifying a Resize mechanism as sketched above. The latter would require the formalism of a *"Well Formed Circuits"* introduced by Choudhury et al [Cho+13], to specify the inclusion of interactive bootstrapping gates in the circuit (see the discussion above §3.1.2). In addition, the implementation, although intuitive (§3.1.2) would be more delicate to describe, since bivariate polynomials would have coefficients in $\mathbb{Z}$ but would be well defined only modulo p.

We firstly require that TAE is a threshold encryption scheme with transparent setup, as recalled in §1.1.1. Namely, we specify an encryption algorithm, which takes as parameter the n public keys that are on the bulletin board. We leave the reader to make the straightforward adaptation in the algorithms, for the cases where up to t keys were not published. Notice that all honest players ($\geq t+1$) do publish their public keys, since they are instructed to, and are always able to do so.

We consider a finite field $\mathbb{F}_p$ of prime order p. In practice, $\mathbb{F}_p$ is the field of the definition of the arithmetic circuit to be computed in MPC. For instance, in the case of an implementation using the ElGamal encryption "in the exponent" as baseline (defined in §B.2), then p is small enough so this baseline (cf §2.4.2) has efficient decryption. We fix E any public key encryption scheme as 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.

Anticipating on the long list below, let us simplify a bit the task of the reader. First, **PrivDec** outputs the plaintext only to a designated recipient. The implementation will be simple from a conceptual perspective: each player applies **PubDec**.Contrib on the ciphertext, then encrypts the output under the recipient's public key. Also, notice that **Add** is just a particular case of **LinComb**, which we single-out for clarity.

Definition 3. A $(t+1)$*-out-of n threshold encryption scheme with transparent setup* over $\mathbb{F}_p$ is the data of a space $\mathcal{C}$ denoted as the *global ciphertext space*, spaces $s\mathcal{K}$ and $p\mathcal{K}$ denoted as the "secret keys" and "public keys" spaces, and of the following algorithms. The $\{0,1\}^*$ denotes binary strings of unspecified lengths, but in our implementation it will be a vector of n elements of the E plaintext or E-ciphertextspace (a row of the total $n \times n$ ciphertext matrix).

KeyGen(): $\emptyset \to (s\mathcal{K}, p\mathcal{K})$

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

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

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

PrivDec. Contrib : $p\mathcal{K} \times s\mathcal{K} \times \mathcal{C} \to \{0,1\}^* \cup$ abort. On input pk_r, which is the recipient's public key, sk_i and c, outputs $E(\mathsf{pk}_r,$ **PubDec**.Contrib$(sk_i, c))$, or abort. Notice that the notation $\{0,1\}^*$ is because the length is unspecified, but in our implementation it will be a vector of n E_{pk_r}-ciphertexts of elements of $\mathbb{F}_p$.

PrivDec. Combine : $(\{0,1\}^*)^{t+1} \to \{0,1\}^* \cup$ abort takes $t+1$ outputs of **PrivDec**.Contrib and outputs in $(\{0,1\}^*)$ or abort. In our implementation, the output will be an array of size $n \times n$, partially filled with E_{pk_r}-ciphertexts of elements of $\mathbb{F}_p$

Add. Contrib : $p\mathcal{K}^n \times s\mathcal{K} \times \mathcal{C}^2\{0,1\}^* \cup$ abort, denoted as *addition share* if not abort.

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

LinComb. Contrib with public parameters $L \in \mathbb{N}^*$ and $(\lambda_1, \ldots, \lambda_L) \in \mathbb{F}_p^L$: $p\mathcal{K}^n \times s\mathcal{K} \times \mathcal{C}^L\{0,1\}^* \cup$ abort.

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

That satisfy *completeness, privacy: IND-CPA and simulatability of decryption shares* as defined below.

3.2.1 Completeness. We firstly introduce the following recursive definition.

Definition 4 (TAE.ciphertext)**.** First, any correctly computed **Encrypt**(x), for all $x \in \mathbb{F}_p$, is a TAE.ciphertext of x. Then, for any TAE.ciphertext $c_m, c_{m'}$, of $m,\ m' \in \mathbb{F}_p$, and any $t+1$ correctly computed addition shares output by distinct players, then, the output of **Add**.Combine on these shares is by definition a TAE.ciphertext of $m+m' \in \mathbb{F}_p$. More generally, for any $t+1$ correctly computed linear combination shares, output by distinct players on the same inputs and parameters, then the **LinComb**.Combine of these shares is by definition a TAE.ciphertext of the linear combination.

Then, the completeness requirement is that for any $m \in \mathbb{F}_p$, then any TAE.ciphertext c_m of m decrypts to m. Namely, m is the output of **PubDec**.Combine applied on any $t+1$ correctly computed decryption shares output by distinct players from **PubDec**.Contrib on input c_m. Likewise, we require that the **PrivDec**.Combine of any $t+1$ correctly computed outputs of **PubDec**.Contrib on input the same TAE.ciphertext c_m, be equal to a of m encrypted with E under the recipient's public key.

3.2.2 IND-CPA Is defined by the following game. Consider a PPT adversary playing with a challenger, who runs $(\mathsf{sk}_i, \mathsf{pk}_i) :=$ KeyGen() $\forall i \in [n]$ and gives all the pk_i to the adversary. Then, the adversary can initially request "corruption" of any index i, up to a total of t corruptions, in the following sense. Upon corruption request for any i_0, the challenger then reveals sk_{i_0} to the adversary. When this happens, the adversary can, in addition, replace pk_{i_0} by one of his choice.

Throughout the game, including after he received the challenge ciphertext below, we also allow the adversary to query the correctly computed output of **PrivDec**.Contrib or **LinComb**.Contrib by any (possibly honest) player, on any inputs in $\mathcal{C}$ of his choice. The only limitation is that in , the recipient is an *uncorrupt* player.

IND-CPA means that, upon submitting two plaintexts m_0, m_1 to the challenger, then being issued c the encryption of one of them, the adversary has a negligible advantage in distinguishing whether c is an encryption of m_0 or m_1.

Further comments on the power of the adversary in the IND-CPA game. Notice that we allowed the adversary to query contributions of **PrivDec**, and **LinComb**. But actually, the adversary and the simulator have no more power than in the definition of threshold homomorphic encryption in [CDN01] (see also Definition 8 in the appendix). Indeed, in [CDN01] the adversary can locally compute the homomorphic linear combinations of any ciphertexts of his choice. Whereas in our definition, computation of homomorphic linear combinations require the contribution of at least one uncorrupt player.

Also, we made the limitation that the recipient of the queries to **PrivDec**.Contrib is an uncorrupt player because, otherwise, this would enable the adversary to obtain the decryption of any ciphertext, i.e. as in IND-CCA, which we do not guarantee.

3.2.3 Simulatability of public decryption shares. Is defined as in appendix §A. Briefly: there exists a simulator that, on input a plaintext m, a correctly computed **Encrypt** c_m of it, and correctly computed decryption shares from a set of t player indices denoted as "corrupt", outputs $n-t$ strings that are computationally indistinguishable from valid decryption shares from the remaining player indices, even for an adversary holding the secret keys of the corrupt indices.

3.3 Implementing TAE

We use the notations of §2.4. We consider a public key encryption scheme (KeyGen, E, Dec) over $\mathbb{F}_p$ as defined in §2.4.2, with the notations that we recall as follows. Let $\perp$ denote the empty value. Given n public keys $\mathsf{pk}_1 \ldots, \mathsf{pk}_n$, denote C_i the corresponding ciphertext space. For brevity, we simplify the encryption notation $E(\mathsf{pk}_i, x)$ to $E_i(x)$. Then, we define the global ciphertext space of the TAE, denoted as $\mathcal{C}$, as the subset of $n \times n$ arrays such that each row i either consists in a vector in $[C_1, \ldots, C_n]$, or, the empty vector $\perp^n$. We now introduce the following intrinsic definition. As stated in Proposition 9, with respect to the following implementation of TAE, this definition will turn out to be synonymous Definition 4 of a TAE.ciphertext. In short, as sketched in §3.1.1, a ciphertext is a $n \times n$ array, of which t rows are empty, while the remaining $t+1$ rows consist of ciphertexts of evaluations of a bivariate symmetric polynomial.

Definition 5. A well formed ciphertext $c \in \mathcal{C}$ is an array such that there exists a bivariate symmetric polynomial $B(X,Y) \in \mathbb{F}_p[X,Y]_{t,t}$, and $t+1$ row indices, denoted $\mathcal{I} \subset [1, \ldots, n]$, such that the entries on these rows are encryptions of evaluations of B:

$$c_{i,j} := E_j(B(\alpha_i, \alpha_j)) \ , \forall i \in \mathcal{I} \ , j \in [n] \tag{1}$$

The other t rows are empty. We say that $c \in \mathcal{C}$ is a well formed ciphertext of plaintext x if $x = B(0,0)$

The first important property of a well formed ciphertext is that, for every fixed column index j, then the nonempty entries on the j-th plaintext column are $t+1$ evaluations of the polynomial $B_j(X) := B(X, \alpha_j)$, which is of degree $t+1$, and thus, by Lagrange interpolation are enough to interpolate the whole polynomial $B_j(X)$.

The second important property of a well formed ciphertext is that, by symmetry of B, we have equality of the plaintexts $B(\alpha_i, \alpha_j) = B(\alpha_j, \alpha_i)$ when the entry (i,j) is nonempty.

3.3.1 ***Encrypt*** Let $(\mathsf{pk} \in p\mathcal{K}^n, m \in \mathbb{F}_p)$ be the inputs.

Sample a random symmetric bivariate polynomial $B(X,Y) \in \mathbb{F}_p[X,Y]_{t,t}$, such that $B(0,0) = m$. Choose any subset of $t+1$ indices $\mathcal{I} \subset [1, \ldots, n]$. Output the $n \times n$ array, with the rows with indices in $\mathcal{I}$ as follows, and the other rows empty: $c_{m,(ij)} := E_j(B(\alpha_i, \alpha_j))$, $\forall i \in \mathcal{I} \ , j \in [n]$,

3.3.2 Threshold decryption **PubDec.**Contrib: Let (sk_j, c) be the inputs. By Definition 5, if c is a well formed ciphertext, then there are at least $t+1$ nonempty entries on column j, and all of them are correctly decryptable. Denote $(d^c_{i,j})_{i \in \mathcal{I}_j}$, these decryptions, where $\mathcal{I}_j$ denotes the set of row indices of these entries.

PubDec.Combine: on input $t+1$ (correct) decryption shares, deduce the unique symmetric polynomial $B[X,Y] \in \mathbb{F}_p[X,Y]_{t,t}$, such that, for all those $t+1$ correct decryption shares, we have: $B(\alpha_i, \alpha_j) = d^c_{i,j} \in \mathbb{F}_p$. Then output $m := B(0,0) \in \mathbb{F}_p$.

PrivDec.Contrib Let $(\mathsf{pk}_r, \mathsf{sk}_j, c)$ be the inputs. By Definition 5, if c is a well formed ciphertext, then there are $t+1$ nonempty rows, of which we denote the indices $\mathcal{I} \subset [1, \ldots, n]$. Denote $(d^c_{i,j})_{i \in \mathcal{I}}$ the decryptions of the $t+1$ entries in column j. Then, output encryption the list of their encryptions with pk_r: $[E_r(d^c_{i,j}) \ , i \in \mathcal{I}]$

PrivDec.Combine Initialize an empty $n \times n$ array. For each correctly checked contribution $[c_{i,j}^{(out)}, i \in \mathcal{I}]$, from P_j, copy the elements of this list at their positions (i, j) in the array. After receiving $t + 1$ contributions , output the array.

3.3.3 Threshold Homomorphic Linear Operations We describe only the threshold addition (**Add**), of which the threshold linear combination **LinComb** is a straightforward generalization.

Add.Contrib: Let (sk_j, c, c') be the inputs of player j. By Definition 5, if c and c' are two well formed ciphertexts, then let $\mathcal{I}$ and $\mathcal{I}'$ be the corresponding sets of $t + 1$ nonempty row indices. For each c and c', compute the decryption of those $t + 1$ nonempty entries on column j, which we denote: $\left(d_{i,j}^{c}\right)_{i\in\mathcal{I}}$ and $\left(d_{i,j}^{c'}\right)_{i\in\mathcal{I}'}$. Then, compute the t missing entries on each of these $(n = 2t + 1)$-sized columns j, by polynomial interpolation. Next, add together these two n-sized columns, into the column denoted as $[d_{i,j}^{c+c'}, i \in [n]]$. Finally, output encryptions of its entries, into the form of a n-sized row vector, namely:

$$c_j^{(out)} := \left[E_i(d_{j,i}^{c+c'}), i \in [n]\right]. \tag{2}$$

Add.Combine: Initiate an empty $n \times n$ array, that is, filled with $\perp$. For each contribution $c_j^{(out)}$, i.e., addition share, from some player P_j, that comes with a correct proof, then copy this contribution, which we recall is a row vector, into the j-th row of the array. After receiving $t + 1$ such (correct) addition shares,output the array computed so far.

A proof that the above satisfies completeness, IND-CPA & shares simulatability is given in §B.1.

3.4 Proof of Theorem 1

We consider the baseline protocol of [BTHN10] reminded in §2.2 and our above toy definition and implementation of TAE.

3.4.1 Protocol We now present the overall protocol, that is further formalized in Appendix B.4. Note that we later introduce improvements for the triples generation (2) and the termination (4) in Sections 4.1 and 4.2 respectively, but they are in *no way* necessary for theorem 1. The structure is the *same* as in [ZBT08] but we modify the baseline by using TAE with *transparent setup* instead of their additively homomorphic encryption with trusted setup. To formalize the input predistribution, we define an ideal functionality $\mathcal{F}_{EncInput}$, that receives the inputs from players and, on request, gives all the inputs it has received, encrypted using TAE. This functionality is further detailed in Appendix B.3, it is trivially implemented by having players broadcast their inputs, along with a non-interactive proof of plaintext knowledge, which we denote "NIZK-PoPK". In §B.3 we implement an example of such a NIZK-PoPK. It is conceptually as simple as opening $O(n^2)$ 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 $O(\log(n))$.

(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 $\mathcal{F}_{PKI}$. Then it obtains all the submitted public keys.

(1) **Inputs distribution**: Players give their inputs to $\mathcal{F}_{EncInput}$, then retrieve all encrypted inputs from $\mathcal{F}_{EncInput}$ (see above, and the implementation in appendix B.3). Thus for every input wire, the players have agreement on the ciphertext $X = \mathsf{TAE}.\mathbf{Encrypt}(x)$ of the input value. Note that theses ciphertexts can possibly be empty strings.

(2) - (3) **Triples generation and Circuit evaluation**: The structure of these steps is *unchanged* compared to [BTHN10], of which we recall the triple generation for convenience in Figure 3 of §B.5.1 (formalized in the $\mathcal{F}_{ZK}$-hybrid model). The difference is that homomorphic linear combinations, which were computed locally on $\mathcal{E}-$ciphertexts, are now replaced by our threshold mechanism. Namely, on input TAE.ciphertexts and scalar coefficients, each player proves to the king that he correctly performed **LinComb**.Contrib on the ciphertexts. Concretely, it sends to $\mathcal{F}_{ZK}$ the secret witnesses necessary to verify the computation: its secret key, the randomness used for encryption. We describe in the appendix B.4.1 the details of the relations proven in the case of our TAE, where for convenience we included in the witness some intermediary steps, such as the player's decrypted share. Upon receiving any $t+1$ such correct contributions, the king computes **LinComb**.Combine on them then sends the result to all slaves. Note that in this first simplified approach, the computations of linear combinations are interactive. We sketched in §3.1.2 how our TAE instantiated with a SHE can enable noninteractive additions.

(4) **Termination**: This stage is unchanged from [BTHN10].

3.4.2 Proof: The purpose of the proof is to demonstrate that there is a simulator **Sim** such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between i) the real execution of the protocol by the players $\mathcal{P} = \{P_1, P_2, ..., P_N\}$ with some of them controlled by a malicious adversary $\mathcal{A}$ and ii) the ideal execution where the players interact with an ideal functionality $\mathcal{F}$ and corrupted players are controlled by the simulator **Sim**. As in [BTHN10], the simulator follows the protocol. The simulation is presented in appendix B.7.1.

For simplicity, we do the proof as if players initialized $\mathcal{F}_{PKI}$ 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 [ZBT08]. It enables the simulator to simulate $\mathcal{F}_{PKI}$, and thus provide fake keys to the adversary on behalf of honest players. We explain in §B.8 how to to switch to a global setup. In a nutshell, as formally explained in B.7.1, the simulator then extracts the decryption shares from the adversary, since it still simulates $\mathcal{F}_{ZK}$. Furthermore, we detail in appendix B.3 how to implement$\mathcal{F}_{EncInput}$ from a broadcast and explicit NIZK.

The simulator **Sim** i) simulates a key setup with $\mathcal{F}_{PKI}$ ii) then plays the whole protocol to the simulated honest players with inputs 0, using the simulated keys. iii) Finally it uses the share simulatability introduced in §3.3 to send decryption shares to the adversary kings, which he accompanies with a fake proof. This is a key difference compared to [BTHN10]. It is able to do this fake proof because it simulates $\mathcal{F}_{ZK}$. This is the only place where the simulator produces a fake proof. We refer the reader to appendix B.6 for more details.

3.5 Complexity analysis

Let c_I (resp c_O, c_M, c_l) the number of inputs (resp outputs, multiplication and linear combination gates). All ZK proofs with transparent setup considered in 2.3.2 have size at most linear in the circut 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.

Input distribution (synchronous) In the first step, each player sends its input to $\mathcal{F}_{EncInput}$. Implemented with a broadcast + NIZK PoPK, this represents $O(c_I n^3)$ elements communicated per player, vs $O(n)$ in [BTHN10].

Triple generation from [BTHN10] (asynchronous) The generation of a triple requires $O(n^2)$ randomizations, which themselves communicates $O(n)$ ciphertexts. A basic optimization presented in [ZBT08, Appendix A.2] allows to reduce the amortized number of randomizations to n. Since each of the n kings need one triple per multiplication stage, the total communication complexity of generating triples is $O(c_M n^5)$.

Circuit evaluation (communication size) Each linera combination gates in the toy model of TAE requires one interaction and each multiplication gates require two decryptions and a linear combination. Each decryption communicates $O(n^3)$ elements. Recall that each stage is handled by each of the n kings, for a total of $O(n^4)$ elements per stage. The evaluation of the circuit thus communicates $O((c_M + c_l)n^4)$ elements. However as sketched in §3.1.2, linear combinations can be made noninteractive, which removes the term in c_l.

Circuit evaluation (latency) The evaluation of a circuit require $O(d_M + d_l)$ consecutive interactions, where d_M and d_l are the depth in terms of multiplications and linear combinations. However as sketched in §3.1.2, linear combinations can be made noninteractive, which brings down the latency to $O(d_M)$.

Termination from [BTHN10] The termination step requires each encrypted circuit output to be jointly decrypted to the king using **PubDec**.Combine, which thus learns the plaintext result z and sends it 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. Each decryption communicates $O(n^3)$ elements and 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 Section 4.2 an extension of the communication model of [BTHN10] presented in §2.2. It enables, with the on-the-fly encrypted randomness generation presented in section 4.3, to achieve proactive security as detailed in section 4.4.

4.1 Proof of theorem 2

We first outline, in §4.1.1, our new triples generation protocol, denoted as *PreProc*, and details in appendix C.1 how that allows all the honest players to terminate the protocol and to output $\frac{t+t'}{2} - t'$ random multiplication triples unknown to the adversary. We then describe in 4.1.2 the main building blocks used to build the protocol. Note that the latter is independent of the threshold additive encryption scheme. It can be instantiated either with the one considered in [CDN01; BTHN10], 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 $\boxdot$). in order to show that the generation of multiplication triples requires a constant number of consecutive interactions, which is independent of the number of players.

4.1.1 Outline of PreProc The triples generation protocol, presented in figure 7, 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 multiplicativity.
(2) **Verifiable transformation of triples** Each player then verifies the correctness of the multiplication triples as detailed in §4.1.2.A and outputs the set $\mathcal{U}$ of the players who broadcasted correct multiplicative triples. Let us denote $|\mathcal{U}| := t + 1 + t'$ their number[2]. 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 §4.1.2.C, on the set of triples broadcasted by players in $\mathcal{U}$ to extract $\frac{t+t'}{2} - t'$ random multiplication triples unknown to $\mathcal{A}$. This mainly uses the triple transformation protocol presented in §4.1.2.B. Noticeably, unlike in [CHP13], this number of triples taken into account is *variable*.

This protocol only involves one broadcast and two interactive decryptions during the extraction.

4.1.2 Main building blocks

A. *Non-interactive Proof of plaintext multiplication:* We first need a protocol that allows a prover P to give a Zero-Knowledge proof of plaintext multiplication ($ZKPoPM$) such that all players agree on the outcome of the proof. A verifier V wants to verify that, considering a triple (X, Y, Z), the third component Z is indeed the product of the first two components (specifically that $Z = \mathcal{E}(x.y)$ with $X = \mathcal{E}(x)$ (resp Y)). More formally, the prover issues a proof a the relation R_{trip} that we formalize in the Appendix B.4.1. This proof can also be constructed from the general circuitry techniques detailed in §B.5, and such as examplified in §B.3, same as for the other relations formalized in §4.1.2 which we used to construct TAE. Recall that these generic techniques consist in exhibiting a commitment, then proving equality between the plaintext and the committed value, then proving the relation on the committed values. For sake of completeness, let us recall that direct proofs also

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

already exist in the literature for this specific purpose of multiplicative relation between encrypted values. For instance: [DJ01, §4.2] "Protocol Multiplication-mod-n^s" for Paillier ciphertexts, and also in general: [Ben+11, Fig 1] for any semi homomorphic encryption scheme, such as Paillier or ElGamal in the exponent (§B.2.2).

B. *Triples transformation:* 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 $TripTrans$ 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 $\mathcal{A}$. $TripTrans$ outputs $t+1+t'$ triples, say $\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']}$ [3], 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 $\mathbb{x}(\alpha_i) = x^{(i)}, \mathbb{y}(\alpha_i) = y^{(i)}$ and $\mathbb{z}(\alpha_i) = z^{(i)}$ 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 [BSFO12]. 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. Details are presented in figure 4.

C. *Randomness extraction:* We present a protocol called $TripExt$ that extracts $\frac{t+t'}{2}-t'$ random triples unknown to $\mathcal{A}$ from a set of $(t+1+t')$ triples. The idea of the protocol is inherited from [CHP13] and summed up as follows: from $t+1+t'$ correct triples, the transformation protocol is executed to obtain three polynomials $\mathbb{x}(.)$, $\mathbb{y}(.)$ and $\mathbb{z}(.)$ of degree $\frac{t+t'}{2}$, $\frac{t+t'}{2}$ and $t+t'$, where $\mathbb{z} = \mathbb{x}(.)\mathbb{y}(.)$ holds. The random outputs, unknown to $\mathcal{A}$, are then extracted as $\{(\mathbb{X}(\beta_j), \mathbb{Y}(\beta_j), \mathbb{Z}(\beta_j))\}_{j\in[\frac{t+t'}{2}-t']}$. Details are presented in C.

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

4.2 Novel Computation Structure

We present the new computation structure to evaluate a circuit introduced in §1.2.2. It is not necessary for Thm 1, but is for proactivity. We detail in Appendix D.6, how to wrap our TAE in this novel computation structure.

4.2.1 Stage, and speedup wrt [BTHN10]

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 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:

$$Inv\_stage: \text{any output of a stage signed by at least } t+1 \text{ players is a correct verified stage output.} \quad (3)$$

This essentially forms a *chain of correctness* from distribution to termination. Note that a player cannot terminate until it knows that all honest parties will also terminate. In [BTHN10], 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.

4.2.2 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

$$\textbf{Verif0ut}.value = combine_{sid}(\{contrib_{sid}(\{X_i\}_i, s_j\}_{j\in\mathcal{S}}). \quad (4)$$

The execution of a Stage for a player is presented in figure 8 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

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

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.

4.3 On-the-fly Generation of Threshold-Additive Encrypted Random Value

We propose a linear threshold construction to produce an encrypted random value without setup. This leverages the construction introduced by Cramer-Damgård-Ishai [CDI05, §4] and denoted *pseudo-random 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 6. *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.*

We give details of Theorem 6 in appendix E 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 4.4 to enable proactive security.

4.4 Proactive security

In §4.4.1 we define the model, denoted as "proactive". We then address the three threats mentionned in the introduction. Namely, we describe in §4.4.2 how to refresh the keys, both for encryption and for the randomness generation (§4.3), then in §4.4.3 how to refresh the plaintext shares constituting the ciphertexts. Note that we also provide further explanations about how our model stands compared to previous works [SLL10] [Bar+14] [Cac+02] in F.1.

4.4.1 Model We define locally, at each player, a monotonically increasing counter denoted as epoch number: $e = 1, 2, \ldots$. Furthermore, we denote that a player is performing a "closing operation" if he is currently participating to one of the sub-protocols, detailed in §4.4.2 and §4.4.3, which consist in refreshing the keys and the ciphertexts. The adversary can corrupt any player at any time, but for each positive number e, then no more than a total of t distinct players can ever be corrupted while they are in epoch e. Furthermore, a player performing a closing operation of some epoch e which is corrupted, counts also a corrupted with respect to epoch $e+1$. Each player has in memory: his secret key relatively of the current epoch, his set of secret keys $(r_A)_{j\in A}$ for the PRSS relatively to the current epoch, and the TAE.ciphertexts on which he is currently operating (as a slave or king, in n simultaneous instances). The adversary learns the memory of every player at the instant when he becomes corrupt, and stores this information forever (even after the player is decorrupted). Thus, to prevent the adversary to gain too much knowledge, we assume that players are able to *erase* all the material not needed anymore of their memories. Let us outline the chronology of a closing

4.4.2 Closing of an epoch (A) The first task is to have all players in epoch e obtain a new public / private key pair, relatively to epoch $e+1$. On the one hand, for the subset of players who still have their keys of epoch e, we have two options to achieve this task. (i) The simple one is to have these players erase all their memory, generate a new key pair and publish the public key on the bulletin board. So this requires a global clock such that, after a timeout, players who did not publish a new key are treated as dishonest. (ii) The more complicated option, which has the advantage not to use the bulletin board, is to perform the Keygen protocol of §E.0.2, 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. On the other hand, freshly decorrupted players have had their memory erased, including their secret key of epoch e, which precludes the second option. Thus, only the first option (i) is available for them.

(B) Next, players generate new PRSS keys. For this they perform $\binom{n}{n-t}$ executions of the Keygen protocol of §E.0.2. Each execution has parameter a set A of $n-t$ recipient players.

(C) Finally, players refresh the ciphertexts which are to be used in future epochs, as sketched in §1.2.4 and detailed below in §4.4.3. Note that output ciphertexts are encrypted with the fresh set of keys. Only then, they can erase from their memory their secret keys of previous epochs, and all plaintexts and ciphertexts related to previous epochs.

4.4.3 Refresh of the (encrypted) shares We recall that a well formed ciphertext c_s of some secret plaintext $s \in \mathbb{F}_p$, is an array of encryptions of evaluations of a symmetric bivariate polynomial $B \in \mathbb{F}_p[X,Y]_{t,t}$ such that $B(0,0) = s$. Our solution is that players collectively generate a ciphertext c_0 of 0, in the form of an array of encryption of evaluations of a random symmetric bivariate polynomial $Q \in \mathbb{F}_p[X,Y]_{t,t}$ such that $Q(0,0) = 0$. Finally, players perform TAE.**Add** of c_s and c_0, which outputs a new ciphertext c'_s of s encrypted with the new keys. In detail, generation of such a Q, which we denote as TAE.**RandZero**, along with summation with c_s, consists in two stages. Firstly each player l generates a random bivariate polynomial $Q^l(X,Y)$ with zero constant coefficient, then sends the array of encryptions $Q^l(\alpha_i, \alpha_j)$ (with exactly $t+1$ nonempty rows) to the king along with a ZK proof that the constant coefficient is

indeed 0, that is, that $Q^l(0,0) = 0$. The output of this first stage is the concatenation of any $t+1$ such valid contributions. Then in the next stage, players execute TAE.**LinComb** to compute the summation of these $t+1$ contributions, which is denoted c_0, along with c_s. The proof of the following lemma in Appendix F.2 ensures the privacy of this refresh.

Lemma 7. *If $\mathcal{A}$ corrupts no more than t parties performing a "closing operation", the view of $\mathcal{A}$ during the refresh operation is distributed independently of the plaintext s and of its view in previous epochs.*

References

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

[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 Application to Logarithmic Transparent Threshold Signatures". In: *accepted to ASIACRYPT 2021*. Cryptology ePrint Archive, Report 2020/1447. 2021.

[Ash+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.

[Bac+14] Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. "Asynchronous MPC with a Strict Honest Majority Using Non-equivocation". In: *PODC*. 2014.

[Bad+20] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. "Secure MPC: Laziness Leads to GOD". In: *ASIACRYPT*. 2020.

[Bar+14] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. "How to Withstand Mobile Virus Attacks, Revisited". In: *ACM PODC*. 2014.

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

[Bea91] Donald Beaver. "Foundations of Secure Interactive Computing". In: *CRYPTO*. 1991.

[Ben+11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. "Semi-homomorphic Encryption and Multiparty Computation". In: *EUROCRYPT*. 2011.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. 2012.

[BOKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. "Asynchronous Secure Computations with Optimal Resilience (Extended Abstract)". In: *ACM PODC*. 1994.

[BS+18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. iacr eprint 2018/046. shorter version in CRYPTO'19. 2018.

[BSFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. "Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority". In: *CRYPTO*. 2012.

[BT99] Fabrice Boudot and Jacques Traoré. "Efficient Publicly Verifiable Secret Sharing Schemes with Fast or Delayed Recovery". In: *Information and Communication Security, Second International Conference, ICICS'99*. 1999.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. "Perfectly-Secure MPC with Linear Communication Complexity". In: *IACR TCC*. 2008.

[BTHN10] Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. "On the theoretical gap between synchronous and asynchronous MPC protocols". In: *ACM PODC*. 2010.

[Cac+02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems". In: *ACM CCS*. 2002.

[Can+02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. "Universally Composable Two-Party and Multi-Party Secure Computation". In: *STOC*. 2002.

[Can04] Ran Canetti. "Universally Composable Signature, Certification, and Authentication". In: *CSFW*. IEEE Computer Society, 2004, p. 219.

[Can+07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. "Universally Composable Security with Global Setup". In: *IACR TCC*. 2007.

[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: *IACR TCC*. 2005.

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

[CFY16] R. K. Cunningham, Benjamin Fuller, and Sophia Yakoubov. "Catching MPC Cheaters: Identification and Openability". In: *Information Theoretic Security*. 2016.

[Cho+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.

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

[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.

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

[CS03] Jan Camenisch and Victor Shoup. "Practical verifiable encryption and decryption of discrete logarithms". In: *CRYPTO*. 2003.

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

[Dam+09] I. Damgård, M. Geisler, M. Kroigaard, and J.B. Nielsen. "Asynchronous multiparty computation: Theory and Implementation". In: *IACR PKC*. 2009.

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

[Das+21] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. *SPURT: Scalable Distributed Randomness Beacon with Transparent Setup*. iacr eprint 2021/100. 2021.

[Daz+08] V. Daza, J. Herranz, P. Morillo, and C. Ràfols. "Ad-Hoc Threshold Broadcast Encryption with Shorter Ciphertexts". In: *Electron. Notes Theor. Comput. Sci.* 192 (2008), pp. 3–15.

[DGLS15] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. "Constant-Round MPC with Fairness and Guarantee of Output Delivery". In: *CRYPTO*. 2015.

[DJ01] Ivan Damgård and Mads Jurik. "A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System". In: *IACR PKC*. 2001.

[DS+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. In: *CRYPTO*. 2001.

[Esc+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits". In: *CRYPTO*. Ed. by Daniele Micciancio and Thomas Ristenpart. 2020.

[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: *IACR PKC*. 2001.

[Gär99] Felix C Gärtner. "Fundamentals of fault-tolerant distributed computing in asynchronous environments". In: *ACM Computing Surveys (CSUR)* (1999).

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. "A Simple BGN-Type Cryptosystem from LWE". In: *EUROCRYPT*. 2010.

[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).

[Gol20] Oded Goldreich. "On (Valiant's) Polynomial-Size Monotone Formula for Majority". In: *Computational Complexity and Property Testing*. 2020.

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

[Her+95] 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 (Extended Abstract)". In: *EUROCRYPT*. 2005.

[HV08] Somayeh Heidarvand and Jorge L. Villar. "Public Verifiability from Pairings in Secret Sharing Schemes". In: *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*. 2008.

[JVS14] Mahabir Prasad Jhanwar, Ayineedi Venkateswarlu, and Reihaneh Safavi-Naini. "Paillier-based publicly verifiable (non-interactive) secret sharing". In: *Designs, Codes and Cryptography* (2014).

[Lai+19] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. "Omniring: Scaling Private Payments Without Trusted Setup". In: *ACM CCS*. 2019.

[Mar+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.

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

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

[Reg09] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *J. ACM* (2009).

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

[RV05] Alexandre Ruiz and Jorge Luis Villar. "Publicly Verfiable Secret Sharing from Paillier's Cryptosystem". In: *WEWoRC 2005 - Western European Workshop on Research in Cryptology, July 5-7, 2005, Leuven, Belgium*. 2005, pp. 98–108.

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

[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.

[TLP20] Georgios Tsimos, Julian Loss, and Charalampos Papamanthou. *Nearly Quadratic Broadcast Without Trusted Setup Under Dishonest Majority*. Cryptology ePrint Archive, Report 2020/894. https://eprint.iacr.org/2020/894. 2020.

[Yin+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: *IACR PKC*. 2001.

[ZBT08] Jesper Buus Nielsen Zuzana Beerliova-Trubiniova Martin Hirt. *Almost-Asynchronous MPC with Faulty Minority*. Cryptology ePrint Archive, Report 2008/416. https://ia.cr/2008/416. 2008.

[ZSR05] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. "APSS: Proactive Secret Sharing in Asynchronous Systems". In: *In preparation* (2005).

A Reminder of Verifiable Threshold Additive Homomorphic Encryption

We first recall the notion of threshold additive homomorphic encryption (AHE), as implemented in [CDN01; BTHN10], at the cost of a *trusted setup*.

Definition 8. (Threshold Additive Homomorphic Encryption)

Let the message space M be a finite group, and let λ be the security parameter. A threshold additive homomorphic cryptosystem on M is a septuplet $(AHE.Setup, AHE.Encrypt, AHE.PartDec, AHE.Combine, AHE.Verify, AHE.Add, AHE.ConstMult)$ of probabilistic, expected polynomial time algorithms, satisfying the following functionalities:

- ***AHE.Setup*** is a randomized procedure that takes as input the number of parties n, a threshold t where $0 \leq t < n$, and a security parameter $\lambda \in \mathbb{Z}$. It outputs a vector $(pk, \mathsf{sk}_1, ..., \mathsf{sk}_n)$ and a verification key vk. We call pk the public key and call sk_i the private key share of party i. Party i is given the private key share (i, sk_i) and uses it to derive a decryption share for a given ciphertext.
- ***AHE.Encrypt*** is a deterministic procedure that returns a ciphertext $c \leftarrow AHE.Encrypt(\mathsf{pk}, x)$ for any plaintext $x \in M$. Let C denotes the ciphertext space. For brevity, let note $c = \mathcal{E}_{\mathsf{pk}}(x)$.
- ***AHE.PartDec*** is a deterministic procedure that returns, on input an element $c \in C$ and one of the n private key share sk_i, an element μ_i denoted as *decryption share*.
- ***AHE.Combine*** is a deterministic procedure that returns, on input $t+1$ decryption shares $\{\mu_1, ..., \mu_{t+1}\}$, an element $x \leftarrow AHE.Combine(\{\mu_1, ..., \mu_{t+1}\})$.
- ***AHE.Verify*** is a deterministic procedure that, on input the public key pk, the verification key vk, a ciphertext c and a decryption share μ, outputs valid or invalid. When the output is valid we say that μ is a valid decryption share of c (and that c is a valid ciphertext).
- ***AHE.Add*** is a deterministic procedure that, on input elements $c_1 \in \mathcal{E}_{\mathsf{pk}}(x_1)$ and $c_2 \in \mathcal{E}_{\mathsf{pk}}(x_2)$, returns an element $c_3 \in \mathcal{E}_{\mathsf{pk}}(x_1 + x_2)$. Let represent *AHE.Add* by $\boxplus$, and note $\mathcal{E}_{\mathsf{pk}}(x_3) = \mathcal{E}_{\mathsf{pk}}(x_1) \boxplus \mathcal{E}_{\mathsf{pk}}(x_2)$.
- ***AHE.Mult*** is a direct extension of *AHE.Add*, that for any integer $a \in \mathbb{Z}_N$ and for a ciphtertext $c \in \mathcal{E}_{pk}(x)$, returns $c' \in \mathcal{E}_{\mathsf{pk}}(a.x)$. Let us write $\mathcal{E}_{\mathsf{pk}}(a.x) = a \boxdot \mathcal{E}_{\mathsf{pk}}(x)$.

and such that we have privacy (IND-CPA and simulatability of decryption shares) and decryption consistency as defined below.

Privacy: IND-CPA Let us introduce the following game between a challenger and a static adversary $\mathcal{A}$. Both are given n, t, and a security parameter $\lambda \in \mathbb{Z}$ as input.

Setup : The challenger runs $AHE.Setup(n, t, \lambda)$ to obtain a random instance $(pk, \mathsf{sk}_1, ..., \mathsf{sk}_n)$. It gives the adversary pk and all sk_j for $j \in S$

Corruption : The adversary outputs a set $S \subset \{1, ..., n\}$ of at most t parties, then receives their secret keys from the challenger.

Challenge : The adversary sends two messages m_0, m_1 of equal length. The challenger picks a random $b \in \{0, 1\}$ and lets $c^b = AHE.Encrypt(\mathsf{pk}, m_b)$. It gives c^b to the adversary.

Guess Algorithm $\mathcal{A}$ outputs its guess $b' \in \{0, 1\}$ for b and wins the game if $b = b'$

The IND-CPA requirement is that the function $AdvCPA_{\mathcal{A},n,t}(\lambda) := \left|Pr[b = b'] - \frac{1}{2}\right|$, denoted as the advantage of $\mathcal{A}$, is negligible in λ.

Privacy: Simultability of decryption shares There exists a PPT simulator *Sim* which, on input a set of indices $\mathcal{I} \subset [n]$ of size at most t, a plaintext m, a correctly computed encryption c_m of it, and any set of valid decryption shares $\{\mu_i,\ i \in \mathcal{I}\}$, produces simulated decryption shares $\{\mu'_i\}_{i \in [n] \setminus \mathcal{I}}$; such that on input: any output $(\mathsf{pk}, \mathsf{sk}_1, ..., \mathsf{sk}_n)$ of AHE.Setup, any set $\mathcal{I} \subset [n]$ of at most t indices, any m, any valid ciphertext c_m that decrypts to m (via AHE.PartDec then Combine) and correctly computed decryption shares $\left\{\mu_i := PartDec(c_m, \mathsf{sk}_i),\ i \in \mathcal{I}\right\}$, then, for any $\left\{\mu_i := PartDec(c_m, \mathsf{sk}_i),\ i \in [n] \setminus \mathcal{I}\right\}$ correctly computed decryption shares for the remaining indices we have that the adversary has a negligible advantage, in λ, in distinguishing between the two distributions

$$\left\{c_m, m, \{\mu_i\}_{i \in \mathcal{I}}, Sim\big(c_m, m, \{\mu_i\}_{i \in \mathcal{I}}\big)\right\} \text{ and } \left\{c_m, m, \{\mu_i\}_{i \in \mathcal{I}}, \{\mu_i\}_{i \in [n] \setminus \mathcal{I}}\right\} \tag{5}$$

Decryption consistency We consider a challenger that runs $AHE.Setup(n, t, \lambda)$ to obtain a random instance $(pk, \mathsf{sk}_1, ..., \mathsf{sk}_n)$., then gives *all* this to the adversary. Then the requirement is that the adversary has negligible probability (in λ) in producing any valid ciphertext c along with two sets of $t+1$ valid decryption shares for c, such that their corresponding decryptions (via PartDec then Combine) plaintexts are different.

B Complements on the proof of Theorem 1

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

Completeness We first show that the correctly computed **Add** of two well formed ciphertexts c and c', corresponding to polynomials B and B', is itself a well formed ciphertext, corresponding to polynomial $B + B'$, and thus with plaintext equal to the sum of the plaintexts $B(0) + B'(0)$. First, notice that, by symmetry of the polynomials B and B', we have that the plaintexts of the row vector output by the **Add**.Contrib of player j, are exactly the evaluations

$\left[(B+B')(\alpha_j,\alpha_i),\ i\in[n]\right]$. Thus, considering the $t+1$ filled rows of the array output by Combine, and switching the indices i and j in the previous formula, we have that the i-th row of this array has its plaintexts which are equal to $\left[(B+B')(\alpha_i,\alpha_j),\ j\in[n]\right]$. Thus by construction and Definition 5, the array output by **Add**.Contrib is a well formed ciphertext of plaintext $(B+B')(0)$.

We do not formalize the similar statement that the **LinComb** of a well formed ciphertext, is a well formed ciphertext of the linear combination. We deduce the following proposition, which concludes the proof of the first requirement of completeness:

Proposition 9. *With respect to the implementations of **Encrypt**, **Add** and more generally **LinComb** above, we have that the property of being a* TAE.ciphertext *(Definition 4) is synonymous of being a* well formed ciphertext.

Proof. In one direction, consider a well formed ciphertext c_m of some $m\in\mathbb{F}_p$. Then by construction of **Encrypt**, we have that c_m is a possible output of **Encrypt**(m). Thus by definition c_m is a TAE.ciphertext.

In the other direction, we have by Definition 5 that any **Encrypt** of any $m\in\mathbb{F}_p$ is a well formed ciphertext. Then, by the considerations above, the outputs of correctly computed **Add** and more generally **LinComb**, when applied on well formed ciphertexts, retain this property. In conclusion, by the recursive Definition 4, any TAE.ciphertext is a well formed ciphertext. □

The second completeness criterion is that the proofs attached to correctly generated Contributions are always accepted, which follows from completeness of the ZK proof system.

Privacy: IND-CPA For privacy we consider for simplicity the idealized model where, in all arrays in $\boldsymbol{\mathcal{C}}$ seen by the adversary, then any entry which is E-ciphertext under an honest public key, can be replaced by $\perp$. First, throughout the game, each time the adversary makes **Add**.Contrib, it is returned an array with $t+1$ empty columns and, on the t columns of which he knew the plaintexts, the encryption of the sum of these columns under the t corrupt public keys. Likewise for **LinComb**.Contrib. For **PrivDec**.Contrib he receives an empty vector $(\perp^n)$. Thus, it could compute what it receives from its requests. Second, denote $\mathcal{J}_A$ the set of indices of the t corrupt players. We have that the challenge ciphertext **Encrypt**(m_b) received by the adversary is by definition an array with exactly $t+1$ nonempty rows, of which we denote $\mathcal{I}$ the set of indices. By our idealized model above, the array, as seen by the adversary, has $t+1$ empty columns. Privacy then follows from the following Lemma 10.

Lemma 10. *Fix $m\in\mathbb{F}_p$, and consider the subset $\mathcal{B}_m$ of polynomials $B\in\mathbb{F}_p[X,Y]_{(t,t)}$ such that $B(0,0)=m$. Consider any subset $\mathcal{J}\subset[n]$ of t column indices and any subset $\mathcal{I}\subset[n]$ of $t+1$ row indices. Then, when the polynomial B varies in $\mathcal{B}_m$ such that the nonzero coefficients are sampled uniformly at random, then the subarray of evaluations $\left\{B(\alpha_i,\alpha_j)_{i\in\mathcal{I},\ j\in\mathcal{J}}\right\}$ varies uniformly at random in a subspace of $\mathbb{F}_p^{(t+1)\times t}$, which is the same for every m.*

Proof. We first have that, (i) for any fixed $m\in\mathbb{F}_p$, then the vector of t evaluations $[B(0,\alpha_j),\ j\in\mathcal{J}]$ varies uniformly when the nonzero coefficients of B vary uniformly at random. This is by invertibility of the Vandermonde determinant. (ii) Next, in each column $j\in\mathcal{J}$, the $t+1$ entries in $\mathcal{I}$ are the evaluations at the $(\alpha_{i\in\mathcal{I}})$ of the polynomial $B(X,\alpha_j)$, which varies uniformly in the set of polynomials of degree at most $t+1$ evaluating to $B(0,\alpha_j)$ at 0. Thus these $t+1$ entries vary uniformly in a hyperplane of $\mathbb{F}_p^{t+1}$ (since a $t\times t$ submatrix has full rank, by invertibility of the Vandermonde determinant) which depends only on the value of $B(0,\alpha_j)$ Combining with (i) concludes the proof of lemma 10. □

Privacy: shares simulatability By proposition 9, since c_m is a TAE.ciphertext, we have both: exactly $t+1$ rows of c are nonempty, of which we denote $\mathcal{I}$ the indices, and, there is a unique symmetric bivariate polynomial $B\in\mathbb{F}_p[X,Y]_{t,t}$ such that the plaintexts on these rows, are equal to evaluations of B. Denote $\mathcal{J}\subset[n]$ the set of the t "corrupt" column indices. The starting point is that the simulator knows the decryption share for each $j\in\mathcal{J}$. By definition, this decryption share is the set of the $t+1$ plaintexts of the nonempty entries of the j-th column of c_m, namely, of the entries on rows in $\mathcal{I}$. They linearly determine the polynomial $B(X,\alpha_j)$. Thus the simulator knows all evaluations $B(\alpha_i,\alpha_j)_{i\in\mathcal{I},j\in\mathcal{J}}$. Thus be symmetry of B, he knows all evaluations $B(\alpha_j,\alpha_i)_{i\in\mathcal{I},j\in\mathcal{J}}$. In particular, for every uncorrupt column index j', he knows t evaluations on it. In order to fully determine the polynomial $B(X,\alpha_{j'})$, and thus all its evaluations on column j', it thus remains to know one more evaluation. But, let us notice that m and the t corrupt decryption shares are $t+1$ evaluations of the degree $t+1$ polynomial $B(0,Y)$. Thus, they linearly determine $B(0,Y)$. Thus, the simulator knows the evaluations at 0 of all polynomials $B(X,\alpha_{j'})$: this provides the missing $(t+1)$-th evaluation, as desired.

Consistency of decryptions By proposition 9, for any TAE.ciphertext c, we have both: exactly $t+1$ rows of c are nonempty, of which we denote $\mathcal{I}$ the indices, and, there is a unique symmetric bivariate polynomial $B\in\mathbb{F}_p[X,Y]_{t,t}$ such that the plaintexts on these rows are equal to evaluations of B. Since both sets of **PubDec**.Contrib are valid, soundness of the ZK proofs guarantees that they are both correct decryptions of the entries on $\mathcal{I}$ of $t+1$ column indices. Thus, they are evaluations of the same symmetric bivariate polynomial B, so in both cases the output of **PubDec**.Combine is the same $B(0)$.

B.2 Two examples of Semi Homomorphic Encryption (SHE) for Noninteractive Additions in TAE

B.2.1 Example: Paillier The Paillier encryption scheme, as e.g., recalled in [Ben+11, §2.1], has plaintext space $\mathbb{Z}/N\mathbb{Z}$ with public key pk $:= N$ a large product of two secret primes, which themselves constitute the secret key, and ciphertext space $(\mathbb{Z}/N^2\mathbb{Z})^*$. For our purpose we need that p be smaller than any such N 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. Decryption in $\mathbb{F}_p$ returns by definition the plaintext output by Paillier decryption if this plaintext is in $[0,\dots,p-1]\subset[0,\dots,N-1]$, else it returns abort.

B.2.2 Example: 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 [Ben+11]. 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 plaintext space equal to $\mathbb{F}_p$. We consider $\mathbb{F}_p$ as the subset $[0, \dots, p-1] \subset \mathbb{F}_q$. Then, in turn, we map $\mathbb{F}_q$ to $\mathbb{G}$ by $x \to x.g$ (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.) If a discrete logarithm $x \in [0, \dots, p-1]$ is found, then output $x \bmod p \in \mathbb{F}_p$. Else, output abort. Thus, to enable decryption, we have another requirement on p to require, which is that p is small enough such that every discrete log of absolute value smaller than p can be efficiently computed.

B.3 Implementation of Input predistribution

$\mathcal{F}_{EncInput}$ is the functionality that receives plaintext inputs from players, and broadcasts to all players a TAE.ciphertext of the received inputs. To implement it is trivial assuming an initial broadcast round and the ideal functionality $\mathcal{F}_{NIZK}$ for noninteractive zero knowledge proofs, as defined in [GOS12]. Namely, players broadcast TAE.ciphertext of their inputs along with a NIZK proof of plaintext knowledge (PoPK).

For completeness, let us give an explicit example of *succint* NIZK PoPK in the case of TAE with el Gamal in-the-exponent encryption. The prover has a secret plaintext $s \in \mathbb{F}_q$. Let $(\mathsf{pk}_i)_i$ be the public keys of the n players. The prover samples $B = \sum \alpha_{k,l} X^k Y^l$ a secret bivariate polynomial of degree at most t, such that $\alpha_{0,0} := s$. He computes the TAE.ciphertext consisting in the $n \times n$ array c_s of the el Gamal in-the-exponent ciphertexts $c_{i,j} = E_{\mathsf{pk}_j}(B(i,j))$ for all $i, j \in [n]$ (actually only $t+1$ nonempty lines out of n are required). The goal of the prover is to prove knowledge of some bivariate polynomial B' of bidegree degree smaller than (t,t), such that $(c_{i,j})_{i,j}$ are encryptions of $B'(i,j)$.

For each i, j, denote $r_{i,j}$ secret random elements of $\mathbb{F}_q$ sampled by the prover to compute them. The key point is that the $n \times n$ el Gamal ciphertexts $c_{i,j}$ are obtained by *linear forms* in the secrets inputs of the prover, namely, the $(r_{i,j})$ and $(\alpha_{k,l})$:

$$(6)\quad E_{\mathsf{pk}_j}(B(i,j)) = \left\{ r_{i,j}.\mathsf{pk}_j \ ,\ \left(\sum_{k,l} \alpha_{k,l} i^k j^l \alpha_{k,l}\right).g + r_{i,j}.g \right\} \ \forall (i,j)$$

The prover is left with computing a public single compact commitment $P = \textsc{Com}\big((\alpha_{k,l})_{k,l \le t}, (r_{i,j})_{i,j \le n}\big)$ and ZK proves that the content of the public P satisfies the $n \times n$ affine forms (6).

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 chosen to be the same group as 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. These elements are generators of $\mathbb{G}$, which, by uniformity of the sampling, 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 $n \times n$ affine forms of (6), 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 $O(n^2)$: 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 $n \times n$ affine forms (6) can be brought down to the one of *one single* linear 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.3.1 Remarks . A direct PoPK (in the univariate case) is described in [Sch99, §3]. However, it is simpler to construct since 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 of B, $\alpha_{k,l}$, are in the small range $[0, \dots, p-1]$, in order to guarantee that decryption of $c_{i,j}$ is tractable. This range proof can be done on the same commitment P, with equally compressed size, using the range proof detailed in [AC20].

For the case of Paillier encryption, in [FS01] they provide a NIZK proof of equality between the plaintext of a Paillier ciphertext, and the opening of a Pedersen commitment. An interesting question would be to compress their proof.

B.4 Protocol details

B.4.1 Relations to be proven in ZK in the protocol

Encrypt

$$R_{Encrypt} = \Big\{ c_m \in \boldsymbol{\mathcal{C}} \ ;\ B(X,Y) := \sum_{i \le j} b_{ij}(X^i Y^j + X^j Y^i) \in \mathbb{F}_p[X,Y]_{\le t,t} \ :$$
$$c_{m,(i,j)} = E_j(B(\alpha_i, \alpha_j)) \ \ \forall i,j \in [n]$$
$$\wedge \ \ b_{ij} \in [0, \dots, p-1] \ \forall i,j \in [n] \Big\}$$

PrivDec

$$R_{PrivDec_r,j} = \Big\{ \mathcal{I}_j \subset \{1,\dots,n\} \text{ of size t+1 } , (c_{i,j})_{i\in\mathcal{I}_j} \in C^{t+1} ,$$
$$\big(c^{(out)}_{(i,j)}\big)_{i=1\dots n} \in C^n \ ; B_j[X] = \sum_{i=0}^{t} b_{i,j}X^i \in \mathbb{F}_p[X]_{\le t} \ ,$$
$$(d^c_{i,j})_{i\in\mathcal{I}_j} \in \mathbb{F}_p^{t+1} \ :$$
$$c_{i,j} \in E(d^c_{i,j}) \ \forall i \in \mathcal{I}_j$$
$$\wedge \ B_j(\alpha_i) = d^c_{i,j} \ \forall i \in \{1,\dots,n\}$$
$$\wedge \ c^{(out)}_{i,j} = E_r(d^c_{i,j}) \ \forall i \in \{1,\dots,n\}\Big\}$$

Add

$$R_{\textbf{Add}} = \Big\{ \mathcal{I}_j \subset \{1,\dots,n\} \text{ of size t+1 } , \mathcal{I}_j^{'} \subset \{1,\dots,n\} \text{ of size t+1 } ,$$
$$(c_{(i,j)})_{i\in\mathcal{I}_j} \in C^{t+1} , (c^{'}_{(i,j)})_{i\in\mathcal{I}_j^{'}} \in C^{t+1} , (c^{(out)}_{(i,j)})_{i=1\dots n} \in C^n \ ;$$
$$B_j[X] = \sum_{i=0}^{t} b_{i,j}X^i \in \mathbb{F}_p[X]_{\le t} \ , \ B_j^{'}[X] = \sum_{i=0}^{t} b^{'}_{i,j}X^i \in \mathbb{F}_p[X]_{\le t} \ ,$$
$$(d^c_{i,j})_{i\in\mathcal{I}_j} \in \mathbb{F}_p^{t+1} \ , \ (d^{c^{'}}_{i,j})_{i\in\mathcal{I}_j^{'}} \in \mathbb{F}_p^{t+1} \ :$$
$$c_{i,j} \in E(d^c_{i,j}) \ \forall i \in \mathcal{I}_j$$
$$\wedge \ c^{'}_{i,j} \in E(d^{c^{'}}_{i,j}) \ \forall i \in \mathcal{I}_j^{'}$$
$$\wedge \ B_j(\alpha_i) = d^c_{i,j} \ \forall i \in \{1,\dots,n\}$$
$$\wedge \ B_j^{'}(\alpha_i) = d^{c^{'}}_{i,j} \ \forall i \in \{1,\dots,n\}$$
$$\wedge \ c^{(out)}_{i,j} = E_i(B_j(\alpha_i) + B_j^{'}(\alpha_i)) \ \forall i \in \{1,\dots,n\}\Big\}$$

MultVerif

$$R_{trip} = \big\{ c_a \in \mathcal{C} \ , \ c_b \in \mathcal{C} \ , \ c_d \in \mathcal{C} \ ;$$
$$A(X,Y) := \sum_{i,j} a_{ij}X^iY^j \in \mathbb{F}_p[X,Y]_{\le t,t} \ ,$$
$$B(X,Y) := \sum_{i,j} b_{ij}X^iY^j \in \mathbb{F}_p[X,Y]_{\le t,t} \ ,$$
$$D(X,Y) := \sum_{i,j} d_{ij}X^iY^j \in \mathbb{F}_p[X,Y]_{\le t,t} \ :$$
$$R_{Encrypt}(a) \ \wedge \ R_{Encrypt}(b) \ \wedge \ R_{Encrypt}(d) \ \wedge \ a.b = d\big\}$$

In more details:

$$R_{trip} = \big\{ c_a \in \mathcal{C} \ , \ c_b \in \mathcal{C} \ , \ c_d \in \mathcal{C} \ ;$$
$$A(X,Y) := \sum_{i\le j} a_{ij}(X^iY^j + X^jY^i) \in \mathbb{F}_p[X,Y]_{\le t,t} \ ,$$
$$B(X,Y) := \sum_{i\le j} b_{ij}(X^iY^j + X^jY^i) \in \mathbb{F}_p[X,Y]_{\le t,t} \ ,$$
$$D(X,Y) := \sum_{i,j} d_{ij}(X^iY^j + X^jY^i) \in \mathbb{F}_p[X,Y]_{\le t,t} \ :$$
$$c_{a,(i,j)} = E_j(A(\alpha_i,\alpha_j)) \ \ \forall i,j \in [n]$$
$$\wedge \ a_{ij} \in [0,...,p-1] \ \forall i,j \in [n]$$
$$\wedge \ c_{b,(i,j)} = E_j(B(\alpha_i,\alpha_j)) \ \ \forall i,j \in [n]$$
$$\wedge \ b_{ij} \in [0,...,p-1] \ \forall i,j \in [n]$$
$$\wedge \ c_{d,(i,j)} = E_j(D(\alpha_i,\alpha_j)) \ \ \forall i,j \in [n]$$
$$\wedge \ d_{ij} \in [0,...,p-1] \ \forall i,j \in [n]$$
$$\wedge \ \ A(0,0).B(0,0) = D(0,0)\big\}$$

B.5 Triple Generation

B.5.1 Triple Generation Protocol from [BTHN10] We detail in figure 3 the triple generation protocol that we use.

B.6 Complement on the proof of the protocol

To prove security, we show that our protocol is secure in the UC model. Specifically, we construct a simulator **Sim** such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between *i*) the real execution $REAL_{\Pi}$ where the players $\mathcal{P} = \{P_1, P_2, ..., P_N\}$ run the protocol Π and the the corrupted players are controlled by a malicious adversary $\mathcal{A}$ and *ii*) the ideal execution $IDEAL_{\mathcal{F}}$ where the players interact with functionality $\mathcal{F}$ and corrupted parties are controlled by the simulator **Sim**. The functionality $\mathcal{F}$ can be seen as a trusted party that handles the entire protocol execution and tells the players what they would output if they executed the protocol correctly. The environment $\mathcal{Z}$ chooses the inputs of the players, may interact with the ideal / real adversary during the execution, and at the end of the execution need to decide whether a real or ideal execution has been taken place. The environment learns the output of the honest players.

Recall the setup assumptions presented in §2.3, namely that there exists functionalities $\mathcal{F}_{PKI}$ and $\mathcal{F}_{ZK}$ that will be used to prove the security of the protocol presented in B.4. In B.5, we give security proofs of the triple generation mechanism before proving in B.7.1 the overall protocol.

B.6.1 Security of triple generation Although there is no agreement among the players on the multiplication triples (A, B, C), we are given certain guarantees about the triples (except with negligible probability):

A. Guarantee 1: If (A, B, C) is accepted triple, then A and B are encryptions of values a and b, and C is an encryption of ab and the adversary is not be able to distinguish a and b from uniformly random values. Thus, to prove security, let us consider an experiment $Triple_{\mathcal{A}}$ between an adversary $\mathcal{A}$ and a challenger defined as:

Protocol *GenTriple* 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 $U \leftarrow E(u), V \leftarrow E(v), X \leftarrow [u \boxdot B_j], Y \leftarrow [v \boxdot A_j]$ and $Z \leftarrow [u \boxdot V]$ and sends (prove, sid, (U, u), (V, v), (X, Y, Z)) to $\mathcal{F}_{ZK}$.
  (2) Requests output from $\mathcal{F}_{ZK}$ until receiving (verification, sid, 1) that proves that : 1) u such that $U \in E(u)$ and $X \in [u \boxdot B_j]$ 2) v such that $V \in E(v)$ and $Y \in [v \boxdot A_j]$ 3) u such that $U \in E(u)$ and $Z \in [u \boxdot V]$.
  (3) Sends $(A_j, B_j, C_j, U, V, X, Y, Z)$ to all parties.
- 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 (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 $(A_0, B_0, C_0) = (E(1, \epsilon), E(1, \epsilon), E(1, \epsilon))$ 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.

Figure 3: Triple generation protocol

(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 parties, and receives their secret keys. Then he executes the triple generation protocol *GenTriple* and gives (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 $Dec(A)$, $Dec(B)$, $Dec(C)$.
(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. 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. □

B. Guarantee 2: When a triple (A, B, C) for a stage sid is accepted, then 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 parties use different randomizers when contributing to different multiplication stages sid. □

C. Guarantee 3: When for the same multiplication stage sid, two honest parties 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.7 Security of multiplication

Lemma 11 (From [BTHN10]). *Assume that party P_i (resp P_j) holds encryptions $X^{(i)}$ and $Y^{(i)}$ (resp $X^{(j)}$ and $Y^{(j)}$), and gets the multiplication triple $(A^{(i)}, B^{(i)}, C^{(i)})$ (resp (j)) from P_k. Then, the multiplication protocol (recalled in Figure 5) leaks no information to the adversary.*

Proof. During the protocol, P_k might learn the decryptions $f^{(i)} = x^{(i)} + a^{(i)} \mod N$ and $g^{(i)} = y^{(i)} + b^{(i)} \mod N$ as well as the decryptions $f^{(j)} = x^{(j)} + a^{(j)} \mod N$ and $g^{(j)} = y^{(j)} + b^{(j)} \mod N$. However, as $X^{(i)}$ and $X^{(j)}$ (resp $Y^{(i)}$ and $Y^{(j)}$) are correct encryptions of x and y per invariant 3, we have $x^{(i)} = x^{(j)} = x$ (resp y). Similarly, $(A^{(i)}, B^{(i)}, C^{(i)})$ (resp (j)) is a correct multiplication triple for a multiplication stage sid. It follows that either 1) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are uniformly random and independent, or 2) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are individually uniformly random and $(a^{(j)}, b^{(j)}) = (a^{(i)}, b^{(i)}) + (\delta_a, \delta_b)$ for (δ_a, δ_b) known to the adversary. In the first case, $(f^{(i)}, g^{(i)})$ and $(f^{(j)}, g^{(j)})$ are uniformly random and independent and thus together leak no information to the adversary. In the second case, $(f^{(i)}, g^{(i)})$ is uniformly random, and therefore leaks no information to the adversary, and $(f^{(j)}, g^{(j)}) = f^{(i)}, g^{(i)}) + (\delta_a, \delta_b)$ and therefore leaks no

more information than $(f^{(i)}, g^{(i)})$ to the adversary, as the adversary can compute it from $(f^{(i)}, g^{(i)})$ in expected poly-time. □

B.7.1 Simulation

A. Simulating the Input distribution The simulator **Sim** starts by simulating $\mathcal{F}_{PKI}$ and generates the cryptographic keys by computing $(pk_i, sk_i) \leftarrow KeyGen(1^\lambda)$ for $i \in [N] \setminus \mathcal{I}$, and define $\mathbf{pk} = (pk_1, ..., pk_n)$, where $\{pk_i\}_{i\in\mathcal{I}}$ are submitted by $\mathcal{A}$ for each corrupted party $P_i (i \in \mathcal{I})$. Next, **Sim** simulates the operations of all honest players in the input distribution phase. Since it does not know the input of the honest players, **Sim** uses $m'_j = 0$ as plaintext for every $j \notin \mathcal{I}$ and compute $c_j = \mathsf{TAE}.Encrypt(\mathbf{pk}, 0)$ with a simulated proof π_j of validity. When the adversary sends a request to $\mathcal{F}_{ZK}$ for $j \notin \mathcal{I}$ on behalf of a corrupted player, **Sim** responds with a confirmation of the validity of the ciphertext c_j. When a corrupted player P_i $(i \in \mathcal{I})$ sends $(\text{prove}, sid, (c_i, \mathbf{pk}), (m_i, r_i))$ to $\mathcal{F}_{ZK}$, **Sim** confirms that indeed $c_i = \mathsf{TAE}.Encrypt(\mathbf{pk}, m_i, r_i)$ and store m_i.

B. Simulating the Computation and Threshold-Decryption Stage In order to simulate the honest parties in this stage, **Sim** proceeds as follows. Initially, **Sim** computes the evaluated ciphertext c based on the input ciphertexts of the input providers. Specifically, **Sim** runs the protocol honestly for additions. Multiplications are evaluated using the Beaver [Bea91] technique recalled in Figure 5. **Sim** runs the protocol honestly for multiplications and, as before, aborts if some share from a corrupted player is not correct. For every $i \in \mathcal{I}$, **Sim** uses sk_i to compute the partial decryption share $p_i = \mathsf{TAE}.\mathbf{PubDec}.\mathsf{Contrib}(sk_i, F)$ (resp G). Next, for every $i \notin \mathcal{I}$, **Sim** simulated the decryption shares of F and G by leveraging the share simulatability of the TAE as explained in §3.3 and therefore computing $p_j = \mathsf{TAE}.SimPubDec(F, \{p_i\}_{i\in\mathcal{I}})$ (resp G) When the adversary sends a request to $\mathcal{F}_{ZK}$ for $j \notin \mathcal{I}$ on behalf of a corrupted player, **Sim** responds with a confirmation of the validity of the partial decryption share p_j. This is called a simulated decryption of F (resp G)

Next, for every $i \in \mathcal{I}$, **Sim** uses sk_i to compute the partial decryption share $p_i = \mathsf{TAE}.\mathbf{PubDec}.\mathsf{Contrib}(sk_i, c)$. Next, considering the output value m, **Sim** leverages the simulatability of the decryption shares of the honest party, and does a simulated decryption of c.

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.

C. 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 $(\mathcal{F}_{PKI}, \mathcal{F}_{ZK})$.

D. The Game $Hyb^1_{\Pi,\mathcal{A},\mathcal{Z}}$ In this game, we modify the real-model experiment in the input distribution stage. Every honest players P_i encrypt they actual inputs $c_i = \mathsf{TAE}.Encrypt(\mathbf{pk}, m_i)$

Claim 11.1. $REAL_{\Pi,\mathcal{A},\mathcal{Z}} \equiv Hyb^1_{\Pi,\mathcal{A},\mathcal{Z}}$

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

E. The Game $Hyb^2_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\mathrm{Hyb}^1_{\Pi,\mathcal{A},\mathcal{Z}}$ except for for the computation of the decryption shares of F and G from honest players during the computation where we do a simulated decryption of F and G to $\mathsf{TAE}.\mathbf{PubDec}.\mathsf{Contrib}(\mathbf{sk}, F)$ and $\mathsf{TAE}.PubDec.\mathsf{Contrib}(\mathbf{sk}, G)$ instead of a doing a simulated decryption of F and G to the random elements f and g.

Claim 11.2. $Hyb^1_{\Pi,\mathcal{A},\mathcal{Z}} \equiv Hyb^2_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. This follows from lemma 11 in which we argued that the multiplication protocol leaks no information to the adversary. □

F. The Game $Hyb^3_{\Pi,\mathcal{A},\mathcal{Z}}$ This game is just like an execution of $\mathrm{Hyb}_{\Pi,\mathcal{A},\mathcal{Z}}2$ except for for the computation of the decryption shares of F and G from honest players during the computation where we do a real decryption instead of a simulated one.

Claim 11.3. $Hyb^2_{\Pi,\mathcal{A},\mathcal{Z}} \equiv Hyb^3_{\Pi,\mathcal{A},\mathcal{Z}}$

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

G. 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 the following difference. Whenever a corrupted party requests output from $\mathcal{F}_{ZK}$ for sid (for $j \notin \mathcal{I}$), the response from $\mathcal{F}_{ZK}$ is (verification, sid, 1), without checking if P_j sent a valid witness.

Claim 11.4. $Hyb^3_{\Pi,\mathcal{A},\mathcal{Z}} \equiv Hyb^4_{\Pi,\mathcal{A},\mathcal{Z}}$

Proof. This follows since in the execution of Π, honest parties always send a valid witness to $\mathcal{F}_{ZK}$, and so the response from $\mathcal{F}_{ZK}$ is the same in both games. □

H. 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 for for the computation of the decryption shares of honest players where we do a real decryption instead of a simulated one.

Claim 11.5. $Hyb^5_{\Pi,\mathcal{A},\mathcal{Z}} \equiv IDEAL_{f,\mathcal{A},\mathcal{Z}}$

Proof. Recall that we are now using the correct inputs m_j on behalf of the honest players P_j $(j \notin \mathcal{I})$. Therefore, the ideal functionality and the protocol are computing on the same input values. Thus, except with negligible probability, c contains the value m returned by the ideal functionality and the simulated output is indistinguishable from the honest decryption of c. □

B.8 Theorem1: Moving to a global setup

The proof of our protocol in 3.4 leverages the fact that $\mathcal{F}_{PKI}$ is initialized at the beginning of the execution, and thus that it can be simulated by **Sim**. This is what enables **Sim** to generates fake keys on behalf of honest players, and thus decrypt the ciphertexts encrypted by the adversary with these keys (just as in [ZBT08, B]). *On the one hand*, removing this assumption is quite simple. Indeed, even if the actual public keys of honest players are imposed to **Sim**, it can anyway extract the plaintexts to the adversary since **Sim** *also* simulates the proof of knowledge functionality $\mathcal{F}_{ZK}$ (and also from $\mathcal{F}_{EncInput}$ or $\mathcal{F}_{NIZK}$, depending is we are in the pre-distributed inputs model or in the initial round of broadcast model). Notice that

Sim actually needs to extract all decryption shares, concretely, the bivariate polynomial in our implementation of TAE. The reason for this precision is that **Sim** needs to maintain the invariant that each ciphertext corresponds to evaluations of a bivariate polynomial of degree $\leq (t, t)$. So this makes non black box use of our implementation (or, we could specify in the protocol that decryption shares should be extractable from the ZK proofs sent, which is the case in the explicit relations that we specify above).

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. [Can+07] 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; Can+07]. Fortunately, lighter primitives than $\mathcal{F}_{ZK}$ are sufficient for MPC (as the ones of [CKS11]).

C Proof of theorem 2

Triples transformation:

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

(1) For each $j \in [\frac{t+t'}{2} + 1]$, the parties locally set $X^{(j)} = A^{(j)}$, $Y^{(j)} = B^{(j)}$, and $Z^{(j)} = C^{(j)}$.
(2) Let the points $\{\alpha_j, x^{(j)}\}_{j\in\in[\frac{t+t'}{2}+1]}$ and the points $\{\alpha_j, y^{(j)}\}_{j\in\in[\frac{t+t'}{2}+1]}$ define the polynomials $\mathbb{x}(.)$ and $\mathbb{y}(.)$ respectively of degree at most $(\frac{t+t'}{2})$.
(3) The parties 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 parties 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 $\{\alpha_j, z^{(j)}\}_{j\in\in[t+1+t']}$ define the polynomial $\mathbb{z}(.)$ of degree at most $t + t'$. The parties output $\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']}$ and terminate.

Figure 4: Triple transforamtion

Lemma 12. *Let $\{(A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[t+1+t']}$ be a set of $t + 1 + t'$ broadcasted triples. Then for every possible adversary $\mathcal{A}$ and every possible scheduler, protocol $TripTrans$ achieves: **(1)** TERMINATION: All the honest parties eventually terminate the protocol **(2)** CORRECTNESS: The protocol outputs $t+1+t'$ triples $\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']}$ 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 $\mathbb{X}(\alpha_i) = \mathcal{E}(\mathbb{x}(\alpha_i)$ for $i \in [t + 1 + t']$ (resp $\mathbb{Y}, \mathbb{Z}$), it holds: $\mathbb{X}(\alpha_i) = X^{(i)}, \mathbb{Y}(\alpha_i) = Y^{(i)}$ and $\mathbb{Z}(\alpha_i) = Z^{(i)}$. (**b**) $\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$ holds iff all the input triples are multiplication triples. **(3)** PRIVACY: If $\mathcal{A}$ knows $t' \leq \frac{t+t'}{2}$ un-encrypted input triples then $\mathcal{A}$ learns t' values on $\mathbb{x}, \mathbb{y}$ and $\mathbb{z}$*

Proof. TERMINATION: This property follows from the termination property of *EncBeaver* (see Lemma 13).

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 $\mathbb{X}(\alpha_i) = X^{(i)}, \mathbb{Y}(\alpha_i) = Y^{(i)}$ and $\mathbb{Z}(\alpha_i) = Z^{(i)}$ holds for $i \in [t + 1 + t']$. To 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 $\mathcal{E}(\mathbb{z}(\alpha_i)) = \mathcal{E}(\mathbb{x}(\alpha_i)\mathbb{y}(\alpha_i))$ 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 $\mathcal{E}(\mathbb{z}(\alpha_i)) = \mathcal{E}(\mathbb{x}(\alpha_i)\mathbb{y}(\alpha_i))$ 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 $\mathcal{E}(\mathbb{z}(\alpha_i)) = \mathcal{E}(\mathbb{x}(\alpha_i)\mathbb{y}(\alpha_i))$ 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.

PRIVACY: First note that if $\mathcal{A}$ knows more than $\frac{t+t'}{2}$ input triples, then it knows all the three polynomials completely. Now to prove the privacy, 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 $z^{(i)}$[5].

□

C.0.1 Proof of EncBeaver

Lemma 13. *For every possible $\mathcal{A}$ and for every possible scheduler, protocol EncBeaver achieves: **(1)** TERMINATION: All the honest parties eventually terminate. **(2)** CORRECTNESS: The protocol outputs $\{E(x^{(j)}.y^{(j)})\}_{j\in[l]}$. **(3)** PRIVACY: The view of $\mathcal{A}$ is distributed independently of the $x^{(j)}$s and $y^{(j)}$s.*

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 $x^{(j)}y^{(j)} = ((x^{(j)} + a^{(j)}) - a^{(j)})((y^{(i)} + b^{(j)}) - b^{(j)}) =$

[5] 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 $G^{(j)} = Y^{(j)} \boxplus B^{(j)}$

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

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

(1) For each $j \in [l]$, each party 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 parties invoke $PubDec(F^{(j)}, G^{(j)})$ to publicly decrypt $\{f^{(l)}, g^{(l)}\}_{j\in[l]}$.
(3) For each $j \in [l]$, the parties 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.

Figure 5: EncBeaver

$f^{(j)}.g^{(j)} + (-f^{(j)}b^{(j)}) + (-g^{(j)}a^{(j)}) + c^{(j)}$. In particular, we have $\mathcal{E}(x^{(j)}y^{(j)}) = \mathcal{E}(f^{(j)}g^{(j)}) \boxplus (-f^{(j)} \boxdot B^{(j)}) \boxplus (-g^{(j)} \boxdot A^{(j)}) \boxplus C^{(j)}$.
PRIVACY: This property is argued as follows: the only step where the parties communicate is during the decryption of $f^{(j)}$ and $g^{(j)}$. Now $f^{(j)} = x^{(j)} - a^{(i)}$ and the fact that $a^{(i)}$ is random and unknown to $\mathcal{A}$ implies that even after learning $f^{(j)}$, the value $x^{(j)}$ remains as secure as it was before from the view point of $\mathcal{A}$. A similar point can be made for $g^{(j)}$. □

Randomness extraction:

Protocol $TripExt(\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']})$

(1) The parties execute the protocol $TripTrans(\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in[t+1+t']}$ and let $\mathbb{x}(.)$, $\mathbb{y}(.)$ and $\mathbb{z}(.)$, respectively of degree $\frac{t+t'}{2}$, $\frac{t+t'}{2}$ and $t + t'$ be the associated polynomials.
(2) The parties compute $\mathbf{A}_i = \mathbb{X}(\beta_i)$,$\mathbf{B}_i = \mathbb{Y}(\beta_i)$ and $\mathbf{C}_i = \mathbb{Z}(\beta_i)$ for $i \in [\frac{t+t'}{2} - t']$ and terminate.

Figure 6: Randomness extraction

Lemma 14. *For every possible $\mathcal{A}$ and for every possible scheduler, protocol $TripExt$ achieves: **(1)** TERMINATION: All the honest parties eventually terminate the protocol **(2)** CORRECTNESS: The $\frac{t+t'}{2} - t'$ output triples $(\mathbf{A}_i = \mathbb{X}(\beta_i)$,$\mathbf{B}_i = \mathbb{Y}(\beta_i)$ and $\mathbf{C}_i = \mathbb{Z}(\beta_i))$ for $i \in [\frac{t+t'}{2} - t']$ are multiplication triples. **(3)** PRIVACY: The view of $\mathcal{A}$ in the protocol is distributed independently of the output multiplication triples $\{(\mathbf{A}_i = \mathbb{X}(\beta_i), \mathbf{B}_i = \mathbb{Y}(\beta_i), \mathbf{C}_i = \mathbb{Z}(\beta_i))\}$ for $i \in [\frac{t+t'}{2} - t']$.*

Proof. TERMINATION: This property directly follows from the termination property of the protocol $TripTrans$.
CORRECTNESS: To argue correctness, we have to show that the triples $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ are valid multiplication triples for $i \in [\frac{t+t'}{2} - t']$. We recall that $(\mathbf{A}_i = \mathbb{X}(\beta_i)$,$\mathbf{B}_i = \mathbb{Y}(\beta_i)$ and $\mathbf{C}_i = \mathbb{Z}(\beta_i))$. To complete the proof, it is enough to show that the protocol ensures the multiplicative relation $\mathcal{E}(\mathbb{z}(.)) = \mathcal{E}(\mathbb{x}(.)\mathbb{y}(.))$ holds. However, this immediately follows from the correctness property of $TripTrans$ and the fact that all the $t + 1 + t'$ input triples are multiplication triples.

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 recall that, by following the privacy property of protocol $TripTrans$, $\mathcal{A}$ learns at most t' points on the polynomials $\mathbb{x}(.)$, $\mathbb{y}(.)$ and $\mathbb{z}(.)$. Specifically, $\mathcal{A}$ knows t' points out of $\{(\alpha_j, x^{(j)})\}_{j\in[t+1+t']}$. Since degree of $\mathbb{x}$ is at most $\frac{t+t'}{2}$, for all choice of A there exist a unique polynomial $\mathbb{x}(.)$ of degree at most $\frac{t+t'}{2}$ which will be consistent with this point ($\mathbb{X}(\gamma) = \mathbf{A}$) and with the prior knowledge of $\mathcal{A}$. Thus, $\mathbb{X}(\beta_i) = \mathbf{A}_i$ will be random to $\mathcal{A}$ for $i \in [\frac{t+t'}{2} - t']$. 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 The Preprocessing phase protocol

Protocol $PreProc$

First synchronous broadcast round

(1) **Triple distribution** - For $i \in [n]$, evey party 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 party P_i executes the following code:
- The parties verify R_{trip} for all triples $(\{(X^{(i)}, Y^{(i)}, Z^{(i)}\}_{i\in[n]})$ and output a set $\mathcal{U}$ consisting of $t + 1 + t'$ parties who broadcast correct triples.

(3) **Triple extraction and termination**- The parties execute the following code:
- The parties execute $TripExt(\{(X^{(j)}, Y^{(j)}, Z^{(j)})\}_{j\in\mathcal{U}})$, output $\frac{t+t'}{2} - t'$ triples $\{(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C})_i\}$ for $i \in [\frac{t+t'}{2} - t']$ and terminate.

Figure 7: Preprocessing overview

Lemma 15. *For every possible $\mathcal{A}$ and every possible scheduler, protocol $PreProc$ achieves: **(1)** TERMINATION: All honest parties terminate the protocol. **(2)** CORRECTNESS: The $\frac{t+t'}{2} - t'$ output triples will be multiplication triples. **(3)** PRIVACY: The $\frac{t+t'}{2} - t'$ output triples are random and unknown to $\mathcal{A}$*

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 parties will terminate the protocol $PreProc$

CORRECTNESS: This property follows from the correctness property of $MultVerif$ and $TripExt$.

PRIVACY: Given that there will be at least $t + 1$ honest parties in set $\mathcal{U}$ and that the multiplication triples broadcasted by the honest parties are random and unknown to $\mathcal{A}$, the privacy property of

$TripExt$ ensures that the output triple in $PreProc$ is random and unknown to $\mathcal{A}$.

□

D Novel computation method 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*** [6] 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*** [6]. 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 9. Each message m is automatically stamped with $kingNb$, the king number that leads the computation. Each message has a type $m.type \in$ {CONTRIBMSG, COMBMSG, VERIFCONTRIB, 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, parties can send four types of messages:

- 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 $sign$.

[6] See D for more details

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 8 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 Optimization

At first glance, it seems that it takes 2 roundtrips for each operation. However, this can be reduced in two ways. First, stages can be linearly combined. For instance, thanks to the properties of our implementation presented in section 3.3, the TAE.**Add** and TAE.**Mult** can be combined into a single stage realizing a linear combination. This can be further combined with a TAE.**PubDec** stage that decrypts the value. Secondly, similarly to what is done in [Yin+19], one can have the player 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 to just one roundtrip.

Remark 16. Our model greatly simplifies the termination phase. A player can halt as soon as he receives the verified final stage output from ***one*** king. Indeed, it carries the signature of $t+1$ players attesting its correctness. By contrast, in [BTHN10], he needs to wait to receive identical signed plaintext outputs from $t+1$ kings before halting.

D.5 Pseudocode of the structure of computation

The protocols are given in Algorithms 11 and 12. 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 9 gives utilities functions used by all parties to execute the protocol and algorithm 10 describes specific functions used by the king.

Lemma 17. *(Verification Phase) For Every possible $\mathcal{A}$ and for every possible scheduler, the Verification Phase achieves:* ***(1)*** *TERMINATION: All honest party will eventually terminate.* ***(2)*** *CORRECTNESS: For an honest king, the phase outputs a Quorum Verification Certificate.*

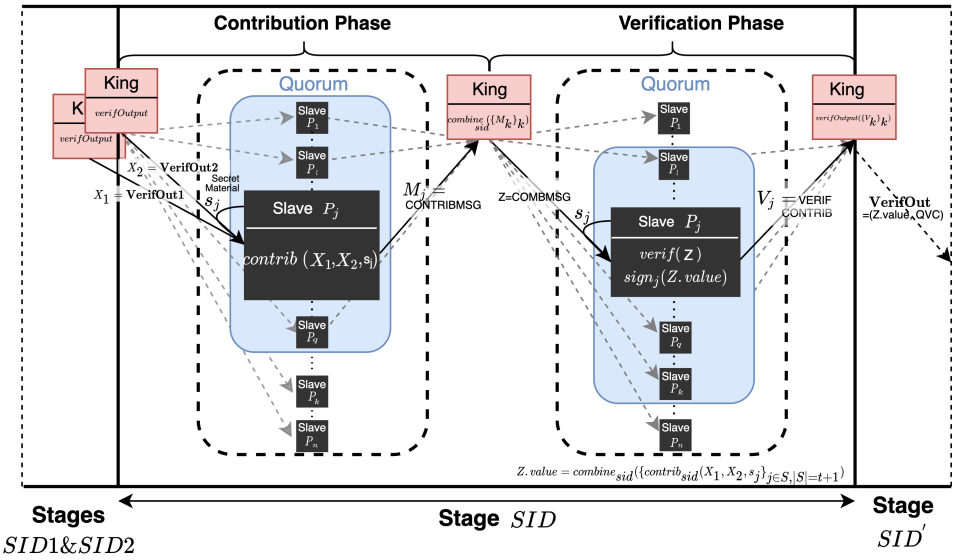

Figure 8: Computation stage for a party P_j. It first receives two *verified stage outputs* X_1 and X_2 from stages SID_1 and SID_2 and uses its secret material s_j to compute its partial contribution using $contrib_{sid}$. The king collects $t+1$ CONTRIBMSG messages with valid proofs, combines the contributions, 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.

Proof. TERMINATION: The honest parties 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 parties, where $C \leq t$, and let further assume C_1 corrupted parties send wrong contributions, C_2 corrupted parties send nothing ever and C_3 corrupted parties send valid contributions, subject to $C_1 + C_2 + C_3 = C$. Since C_2 parties 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 parties. Eventually, all honest parties will receive a *Quorum Verification Certificate*. □

Lemma 18. *(Contribution Phase) For Every possible $\mathcal{A}$ and for every possible scheduler, the Contribution Phase achieves* ***(1)*** *TERMINATION: All honest party will eventually terminate.* ***(2)*** *CORRECTNESS:The phase outputs a Combine Proof*

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

D.6 Using TAE in our computation stages framework

We compile the specification of a TAE, of Definition 3, into a collection of *stages*. We denote this collection of stages as a "MPC-friendly TAE", not to confuse it with a plain TAE, which is a collection of algorithms running locally. In detail, a "MPC-friendly" TAE over $\mathbb{F}_p$ is the data of: a space $\mathcal{C}$ that we denote as the *global ciphertext space*, and of a collection of *stages*, each of them producing ***verified stage outputs***, such that they enjoy the following properties. In the present case where the value associated with such a ***verified stage output*** is a TAE.ciphertext, we call this output a ***verified*** TAE.ciphertext.

- TAE.***Input*** $p\mathcal{K}^n \times \mathcal{C}^* \times \Pi^* \to \{(\mathcal{C}\times\Pi)^n\}$ is a stage that takes as inputs the ciphertexts broadcasted in the first round, and returns a list of n ***verified*** TAE.ciphertext with guarantees that: (a) all kings have the same encrypted plaintexts and (b) for each player P_i who broadcasted $c_{m_i} = \textbf{Encrypt}(\text{pk}, m_i)$ with a valid ZK proof of correct encryption during the initial round, then c_{m_i} is in the output list at index i. Note that this stage has not Contrib function. For the Combine function, the king simply takes the broadcasted TAE.ciphertexts with valid proof and fills an initially empty $n-$dimensional vector. For player indices j that have not broadcast: the king writes $c_j := \textbf{Encrypt}(\text{pk}, 0)$ in the vector box j, and adds a proof of correct encryption.
- TAE.***PubDec*** $s\mathcal{K}^n \times \mathcal{C} \to \mathbb{F}_p$ is a stage that takes as input a ***verified*** TAE.ciphertext c and produces a verified plaintext m such that $m \leftarrow \text{TAE.}\textbf{PubDec}(\textbf{Encrypt}(\text{pk}, m))$.

Let $m \in \mathbb{F}_p$ be a plaintext and let c. We say that c is a well formed TAE ciphertext of $m \in \mathbb{F}_p$ if $m = \text{TAE.}\textbf{PubDec}(c)$. We illustrate how **Add** can be wrapped in interactive stages, that produce ciphertexts signed as valid by $t+1$ players. We have the straightforward generalization to a TAE.**LinComb** stage. Of course one could be more efficient and pack in one single stage, e.g., a linear combination followed by a private opening.

- TAE.***PrivDec*** $s\mathcal{K}^n \times p\mathcal{K} \times \mathcal{C} \to C^*$ is a stage that takes as input a verified TAE.ciphertext c_m and a designated player

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

Figure 9: Utilities

P_r, and produces a ciphertext $E(pk_r, m)$ under the public key pk_r of P_r, such that c_m is a well formed ciphertext of m.

- TAE.***Add*** $p\mathcal{K}^n \times s\mathcal{K}^n \times \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is a stage that takes as inputs two ***verified*** TAE.ciphertexts c_m, of some $m \in \mathbb{F}_p$, and $c_{m'}$, of some $m' \in \mathbb{F}_p$, and produces a *verified* TAE.ciphertext $c_{m+m'}$. It is such that $c_{m+m'}$ is a well formed ciphertext of $m + m'$.

E 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 §E.0.1. We then show in §E.0.2 that this construction makes possible the generation of pairs of public/private keys as well as proactive security. Finally, in §E.0.3, we detail an implementation in our computation structure.

Let first define $F_{kg} : Sk \rightarrow K$ that goes from a private key space Sk to a a public key space K, as a generic function that derives a

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

Figure 10: 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 $MSG(\text{VERIFIED-OUTPUT}, m.sid, out, qvc.sig, \perp)$
(9) **as** a slave:
(10) Upon receiving a COMBMSG message m:
(11) if $verify(m)$ is True:
(12) Send to king $sign(m, s_j)$

Figure 11: 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 $contrib_{sid}(\{m\}, s_j)$

Figure 12: Contribution Phase

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}.

E.0.1 Encrypted Randomness Generator We define a stage, denoted TAE.**Rand**, which has a specification close to a Threshold Coin, as introduced in [CKS05, §4.3.]. Each TAE.**Rand** stage is parametrized by a public coin number, which is encoded in the SID, and takes as public inputs a vector pk of public keys. It outputs a ***verified*** 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 stages with different Kings executing TAE.**Rand** on the same coin number, output a ciphertext of the same value r.

First implementation using broadcast This can be easily implemented during the initial synchronous broadcast round by letting every party sharing a random value; the sum of the shared random values will be common and random to every party. Our goal is to go beyond this naive idea and to propose a randomness generator that works in an asynchronous network.

On-the-fly encrypted randomness generator without broadcast To build TAE.**Rand** without broadcast, we leverage the construction introduced by Cramer-Damgård-Ishai [CDI05, §4] and denoted *pseudorandom secret sharing* (PRSS). It enables players to generate, without interaction, an unlimited number of shared unpredictable random values. They come in the form of Shamir shares, that players generate locally.

We recall in E.1 the PRSS construction, that we enrich with, 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 $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$. 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 **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 E.2 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 §E.0.3.

E.0.2 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

E.0.3 Implementing TAE.***Rand*** The initial step is to implement the trusted dealer of PRSS keys, by the distributed key generation protocol of §E.0.2. The calls to TAE.**Rand** required in this initial step, can either be implemented with the broadcast, or, recursively, from previous calls to TAE.**Rand** with the broadcast-free implementation that we are describing now.
TAE.***Rand*** comes as two consecutive stages. The first one takes no input. The second one outputs a 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 stage takes as parameters a fresh seed a. To be concrete, notice that, in the implementation sketched above in §E.0.1, then a comes as a set of $(t+1)(t+2)/2$ fresh distinct seeds $a_{i,j} \in \mathcal{S}$. Its contribution function is as follows: each player outputs **Encrypt**$(PRSS(j, a))$, along with a proof of correctness, as specfied in E.0.1. Its combination function simply takes as input a set of contributions issued by any set $\mathcal{L} \subset \{1, \ldots, n\}$ of $t+1$ distinct players: $\{(c^l, \pi^l), l \in \mathcal{L}\}$, such that the proofs of correctness π^l are verified, and outputs the concatenation of them along with the proofs.

The second stage 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

$$c_{s(a)} := \text{TAE.}\mathbf{LinComb}_{(\lambda_l)_{l\in\mathcal{L}}}\left(\{c^l\}_{l\in\mathcal{L}}\right). \tag{7}$$

Proposition 19. *The output of these two consecutive stages has the unpredictability property defined as in the game below.*

A proof of proposition 19 is given in E.3.

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 parties n is indeed expected to be small.

E.1 Reminder of Pseudorandom Secret Sharing (PRSS)

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

$$PRSS(l, a) := \sum_{|A|=n-t,\ l \in A} \psi_{r_A}(a) \cdot f_A(l) \tag{8}$$

Where f_A is a fixed public polynomial that we do not specify. Then, by Lemma 3 $PRSS(a)$ is *linearly* reconstructible from any $t + 1$ shares.

E.2 PRSS Implementation

In this implementation, the new space of seeds is $\mathcal{S}^{(t+1)(t+2)/2}$. Consider a player l, with inputs its set of secret keys $(r_A)_{l \in A}$, a seed a and $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ the set of public keys. P_l computes $b^l_{i,j} := PRSS(l, a_{ij})$ on $(t + 1)(t + 2)/2$ fixed public distinct seeds: $a_{i,j} \in \mathcal{S}$, they are the $(t + 1)(t + 2)/2$ coefficients of a symmetric bivariate polynomial $B^l(X, Y) \in \mathbb{F}_p[X, Y]_{(t,t)}$. Second, it computes the the array of its evaluations, on the (α_i, α_j) for $i, j \in [n]^2$, then encrypts the entries in each column j 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 $PRSS(l, a_{ij})$, evaluation at the (α_i, α_j), followed by correct encryption.

E.3 Proof of proposition 19

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 (7). 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 (7) 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 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 O'. On every corruption request for an index j from $\mathcal{A}$, $\mathcal{A}'$ forwards it to O'. Then on response of 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 O' two challenge plaintexts: $m_0 := a$, and any $m_1 \in \mathbb{F}_p$ distinct from m_0.

Then 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: $\mathbf{Encrypt}(PRSS(j, a))_{j \in \mathcal{J}}$, and queries **LinComb** on c_b and these t ciphertexts, with the Lagrange coefficients, to deduce $t+1$ prospective uncorrupt encryptions of PRSS shares: $\overline{PRSS(j, a)}_{j \in [n] \setminus \mathcal{J}}$, 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 O', and otherwise he outputs $b := 1$ to O'.

F Proactive Security

F.1 Model

F.1.1 Similarities with [Bar+14] The model of [Bar+14], 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 "closing". Since there is no global clock in our asynchronous model, it makes no more sense to say that players are together doing a "closing". This is why we defined "closing" relatively to each player. The common point with [Bar+14], is that a player corrupted while performing a "closing" 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 columns in clear of all ciphertexts relative to epoch e, and also has his secret decryption key relatively to epoch $e + 1$.

F.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 §4.4.2, 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. Hence, they enable efficient ZK proof systems, as required by our implementation of 3.3, of whom we sketch an efficient instantiation in §B.3.

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." By contrast, our specification the "closing", which includes the implementation §4.4.3, takes a (small) constant number of stages.

F.1.3 Differences with Cachin-Kursawe-Lysyanskaya-Strobl [Cac+02]

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[7]. So this is orthogonal with our specification of TAE, which is a public key encryption mechanism, such that the adversary sees every TAE.ciphertext sent on the network. There is a second reason for which such a hardware assumption is incompatible with TAE. Indeed, TAE requires players to produce complex ZK proofs of statements that combine, e.g., correct encryption with polynomial evaluations. Players would not be able to produce such ZK proofs if the witness, which is the secret key corresponding to their public key, was concealed in a hardware.

The second difference is that they assume that the adversary obeys the constraint 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."). By contrast, we need not make this delivery assumption within an epoch. Indeed, our closing requires players to stay locally in an epoch, until they have obtained their new keys and all refreshed ciphertexts relatively to the next epoch. Thus, since the $t+1$ honest players obey this rule, they are collectively able to continue computing on ciphertexts (then decrypt the output).

[7] "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."

F.2 Proof of lemma 7

Proof. We consider a well formed ciphertext c_s of some secret plaintext $s \in \mathbb{F}_p$ relatively to some epoch e, and denote B the underlying bivariate polynomial. We denote $\mathcal{I}$ the set of the $t+1$ indices of the nonempty rows of c'_s, and $\mathcal{J}_A$ the set of indices of the at most t corrupt players in epoch $e+1$. During the closing, $\mathcal{A}$ receives an array of E-ciphertexts of evaluations of $B+Q$ on the rows $\mathcal{I}$. We make the same idealized assumption on E as in the proof §B.1 of privacy of our implementation of TAE. Namely, we consider that the adversary received exactly the $(t+1)\times t$ plaintext evaluations of $B' := B+Q$ at $\{\alpha_i, \alpha_j\}_{i\in\mathcal{I}, j\in\mathcal{J}}$ while the columns with indices $[n]\backslash\mathcal{J}_A$ can be considered as empty.

Now, since at least one honest player contributed to Q (with an additive contribution Q^l), we have that the nonzero coefficients of $B' := B+Q$ vary uniformly at random, independently of the coefficients of B. Thus by lemma 10 applied to $m := 0$, the subarray of plaintext evaluations of $B' := B+Q$ at $\{\alpha_i, \alpha_j\}_{i\in\mathcal{I}, j\in\mathcal{J}}$, varies uniformly in a subspace of $\mathbb{F}_p^{(t+1)\times t}$, independently of the subarray of evaluations of B at the same points. □