Almost-Asynchronous MPC under Honest Majority, Revisited

MATTHIEU RAMBAUD and ANTOINE URBAN, Télécom Paris and Institut polytechnique de Paris, France

Multiparty computation does not tolerate $n/3$ corruptions under a plain asynchronous communication network, whatever the computational assumptions. However, Beerliová-Hirt-Nielsen [BHN10, Podc'10] showed that, assuming access to a synchronous broadcast at the beginning of the protocol, enables to tolerate up to $t < n/2$ corruptions. This model is denoted as "Almost asynchronous" MPC. Yet, [BHN10] suffers from limitations: (i) *Setup assumptions:* their protocol is based on an encryption scheme, with homomorphic additivity, such that the secret keys of players are given by a trusted entity ahead of the protocol. It was left as an open question in [BHN10] whether one can remove this assumption, denoted as "trusted setup". (ii) *Common Randomness generation:* the generation of common random secrets 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 a mechanism by which players refresh their (shares of) keys, without the intervention of a trusted entity, with *on the fly* randomness generation. (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, our novel computation framework 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 controled by an adversary $\mathcal{A}$ which we assume *computationally bounded*. MPC protocols in the *synchronous* model are extensively studied ([BH08; Esc+20]). The underlying assumption there is that the delay of the messages in the network is bounded by a *known* constant. However, safety of these protocols fails when this assumption is not satisfied. Thus, protocols [Gär99; Dam+09; HNP05b; CHP13; Bac+14] were developed for the *asynchronous* communication model. This setting comes with limitations: Ben-Or, Kelmer, and Rabin [BKR94] proved that AMPC protocols are possible if and only if $t < n/3$, while we can tolerate $t < n/2$ in a synchronous environment. Moreover, Canetti [Can96] showed that it is impossible to enforce *input provision* which obviously, can represent an important setback for practical applications.

In [BHN10], 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. In their protocol, the circuit is evaluated using the *King/Slaves* paradigm [HNP05b], 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, then, in each instance, 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 terminate 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. It was left as an open problem how to remove this assumption, denoted as "trusted setup": in [BHN10, §4.3 *"our protocol requires quite strong setup assumptions, and it is not clear whether they are necessary."*]. The main contribution of this paper is to remove it. Namely, we assume only the mainstream model of a bulletin-board of public keys ([BCG20; TLP20]), where each player can publish any public key of his choice ahead of the execution. Such loose setup is also known as "transparent", or "ad hoc" [Daz+08; RSY18].

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 plain model of a bulletin-board of public keys, and assuming access to one round of synchronous broadcast at the beginning of the execution, then, any arithmetic circuit over any (small) finite field $\mathbb{F}_p$ can be securely computed, with input provision.*

In what follows (§1.1 and §1.1.5) 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: is 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 parameter 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). We then introduce our new computation framework in §1.1.5. In §2.1 we detail the model, in §2.2 we recall basic cryptographic primitives, in §2.3 we recall the baseline protocol of [BHN10]. In §3 we detail our computation framework. In §4 we specify and implement TAE, in §4.5 we wrap TAE in the previous framework, then in §4.5 we deduce the proof of Theorem 1 by recasting the baseline protocol with these new framework and ingredients.

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

1.1.1 Previous Work and Threshold Encryption with Transparent Setup. Let us briefly recall what is a verifiable threshold encryption scheme. It is a public key cryptosystem between n fixed players, that comes with an algorithm that enables any of these players, on input a ciphertext, to ouputs a "decryption share" along with a ZK proof of correctness. Then, any $t + 1$ decryption shares are enough to efficiently reconstruct the plaintext. This is typically implemented with a trusted dealer of decryption keys, e.g., [CDN01; Cho+13]. How to implement this with a transparent setup follows from a well-known 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] [RSY18, 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 [BHN10], we need to support homomorphic additions on the ciphertexts. The straightforward idea to achieve this is to instantiate the previous scheme (PVSS), with additive encryption. Unfortunately,

in all of the previous schemes, the ones instantiated with additive encryption schemes, support only a limited number of additions. For instance in [Sch99, §5], the PVSS applied to electronic voting, uses, as baseline, the additive variant of el-Gamal that we denote as *in the exponent* (see §2.2.3 below). Thus, decryption is performed by brute-force computation (which is denoted by "can be computed efficiently" in loc. cit., bottom of page 11).

The same limitation is stressed in the Paillier-based [RSY18, Appendix E.2] where it is said that *"it supports a limited number (currently set to n) of homomorphic additions"*. The reason is that, players have *different* plaintext spaces $\mathbb{Z}/N_i\mathbb{Z}$, where the N_i are their public keys. One could consider a common plaintext space $M := [0, \ldots, N-1]$ where $N \leq \min(N_i)$. But then, it makes *no sense* to speak about additions modulo N. Thus, homomorphic additions of PVSS are guaranteed to decrypt correctly only if the plaintexts' sizes remain smaller than $N/2$. Thus this limits the number of additions.

Important Remark. By contrast, let us stress that, assuming a trusted setup, then this issue *does not* occur. For instance in the Paillier additive threshold scheme considered in [CDN01; BHN10], then all plaintexts belong to a fixed $\mathbb{Z}/N\mathbb{Z}$ with *unique* N. Thus, homomorphic additions operate in this single ring of plaintexts. In particular they *do* compute addition modulo N of the plaintexts, so there is no limit on the number of additions before correct decryption.

Remark: Formalization of The Maximum Number of Additions wrt to the Size of the Plaintext. Bendlin et al.[Ben+11, Section 2] coined the notion of "Semi Homomorphic Encryption" (SHE), to denote any public key encryption scheme, not necessarily threshold, that supports a limited number of additions. They formalized this limitation in terms of the size of the plaintext (and also of the randomness, in the cases of [Reg09] and [GHV10]), which grows with the number of consecutive additions performed. This includes Paillier, Regev's LWE based cryptosystem [Reg09] or Gentry, Halevi and Vaikuntanathan's scheme [GHV10]. To this list we add what we denote as el-Gamal in the exponent (see §2.2.3).

1.1.3 *Technical Hurdles with Reducing the size of Plaintexts under Asynchrony.*

To overcome this issue with growth of the size, one could think of mechanisms that enable players to, collectively, reduce the size of the plaintexts when they become too large. We now show that this fails when applied to PVSS and using known techniques, because of our demanding model combining asynchrony and transparent setup.

First attempt. One could think of the following naive mechanism for reducing the sizes of the plaintexts contained in a PVSS. At regular intervals, each honest player would decrypt his share of the PVSS, reduce it modulo $\mathbb{F}_p$ to reduce the size of the plaintext, then rencrypt it with his public key, and send it to the other players. This protocol however does not work in our demanding model. First, a honest player (even up to t of them) could be offline for a long time, while many homomorphic additions are performed in the while. Thus, the size of the plaintext of his shares of secrets has grown very large. Thus when he is back in the protocol, he is are unable to correctly decrypt his shares of secrets. But, under our sharp adversary bound $n = 2t + 1$, the plaintext shares of all honest players are necessary to decrypt the final output.

Second attempt. To reduce the sizes of the plaintexts, one could also think of using the interactive mechanism, proposed by Choudhury-Loftus-Orsini-Patra-Smart, under the name "refresh" [Cho+13]. Like us they consider MPC over a small finite field $\mathbb{F}_p$. But in their case $\mathbb{F}_p$ is embedded in the *single* large plaintext space $\mathbb{Z}/N\mathbb{Z}$, of a threshold additive encryption scheme. The main idea is to use the following "masking" technique in three steps. Players start with common input a ciphertext c_z, such that the plaintext $z \in \mathbb{Z}/N\mathbb{Z}$ is of large size. First players collectively generate a ciphertext c_m of a random value $m \in \mathbb{Z}/N\mathbb{Z}$, denoted the "mask", by using an initial round of all-to-all broadcast of

random ciphertexts c_{m_i} that are summed together. Players then compute the homomorphic sum $c_m + c_z$ and collectively decrypt it to obtain the plaintext $m + z \pmod p$. Finally, they deterministically re-encrypt $m + z$ into c'_{m+z}. In particular, the size of the plaintext of c'_{m+z}, which is $m + z \pmod p$, has become small again. Finally, they homomorphically subtracting c_m to deduce c'_z, which is also a ciphertext of $z \pmod p$, but with smaller plaintext size.

However this approach, which works in the trusted setup context of [Cho+13], *fails* when using the aforementioned previous threshold encryption schemes without a dealer, denoted as PVSS. Recall that a PVSS ciphertext, in all these previous conceptions, consists in the vector of the n encryptions of the same value, encrypted under the n public keys of the players. But first, when instantiating a PVSS with Paillier encryption, then the above "refresh"mechanism makes no sense since the plaintext spaces $\mathbb{Z}/N_i\mathbb{Z}$ of players are *different*. Namely, uniform sampling of a plaintext m makes no sense here. This is the big difference with a threshold Paillier with a trusted setup ([BHN10] [CDN01] [Cho+13]), where there is a *single* common plaintext space.

Second, when instantiating a PVSS with, instead, ElGamal-in-the-exponent ([Sch99, §5], or §2.2.3 below), then we have another problem when trying to apply this above "refresh" mechanism of [Cho+13]. Let consider the small plaintext space $\mathbb{F}_p$ seen as $[0, .., p-1]$ embedded in $\mathbb{Z}/q\mathbb{Z}$. Therefore, sampling a mask m in the whole $[0, ..q-1]$, as required by [Cho+13], would result in a $m + z$ that varies uniformly in the large $[0, \ldots, q-1]$. Therefore, decrypting $m + z$, as required by [Cho+13], would be intractable by brute force.

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 share of *every* player in the PVSS 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 §4.3 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 contain enough material to enable the t remaining players to reconstruct their share of the sum, such that these share also has *small* plaintext sizes. This is what overcomes the previous issue of plaintext size growth encountered in the first attempt above, thus enables unlimited additions.

We specify and implement this mechanism in Section 4. Notice that the notion of threshold encryption is well-know, but the important point is that here, we explicitly require that *the setup is transparent*. Namely, we specify only an algorithm KeyGen(), which each player executes locally, i.e., without interaction, to generate a key pair. Each player must then publish on the bulletin board, ahead of the MPC protocol, the public key that he generated.

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. The central idea is thus that, by having $t + 1$ players add modulo p the plaintexts of their columns of ciphertexts, then by symmetry, they are 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.

Finally, to enforce verifiability, and thus active security (see §1.1.5), our protocol requires non-interactive zero-knowledge proofs (NIZK). This step is non trivial, since the statements to be proven are composed of several inter-dependent predicates. For instance, our **Add** requires the combination of proofs of: correct decryption, interpolation, reduction modulo p, addition, then re-encryption, of evaluations of a polynomial that must be kept secret. Fortunately this is made possible by the recent framework of Attema-Cramer [AC20]. It is indeed *modular*, in the sense that, it enables the prover to prove statements on values of which he separately exhibits a public commitment. Combining with, respectively, DDH-based and RSA-based commitments, we instantiate our scheme, along with these ZK proofs, from el Gamal (in the exponent) and from Paillier encryption. To be complete, let us mention that specific proofs do exist ([CDN01], [FS01, §4] (range proof), [DJ01, §4.2] (multiplicative relation), [Ben+11, Fig 1]) that apply directly to plaintexts encrypted with Paillier. But, they do not enable to prove the composite predicates which we require.

1.1.5 Computation method. Last but not least, we put all these contributions in a novel computation framework suited for asynchronous MPC with honest majority. We abstract out the structure of computation of [BHN10], as follows, which defines our baseline. The circuit is evaluated using the *King/Slaves* paradigm [HNP05a], 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. 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 §3.1.1. This is the main difference with [BHN10]. We motivate this choice of intermediary checking for two reasons. First, it simplifies the termination process. Unlike in [BHN10], 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 7.

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 [BHN10] at a cost of a number of consecutive interactions linear in the number of players. We bring this latency down from linear to a small constant, by leveraging the initial round of synchronous broadcast and an innovative method from Choudhury-Hirt-Patra [CHP13, DISC13], 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 input triples. We enrich this protocol by adding verifiability thanks to Zero-Knowledge proofs. This allows us to make structural modifications to the protocol which have the result of increasing the number of triples generated. In detail, we denote t' this new adjustable parameter presented in section 5. This increases the available degree of freedom and enables to improve resiliency to $t < n/2$, whatever the contributions t' of the adversary, and *without* consensus.

Theorem 2. *(Informal) Assuming $n = 2t + 1$ players in an asynchronous communication network, of which t are maliciously corrupted by a polynomial adversary, in the plain model of a bulletin-board of public keys, and assuming access to one round of synchronous broadcast at the beginning of the execution, then, the players can produce random multiplication triples unknown to the adversary , in a fix (constant) number of consecutive interactions.*

1.2.2 On-the-fly encrypted randomness generation. The generation of a common random encrypted secret was proposed in [BHN10]. It naively consists in asking each player to generate a random element, broadcast an additive encryption of it in the first round, which are then summed. We remove the dependency on the broadcast. Our protocol, described in section 6 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, makes 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].

1.2.3 Proactive security. Ostrovsky and Yung [OY91, Podc'91] 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 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. However, re-encrypting the $((t + 1) \times n)$ shares 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. To prevent this second threat, we detail in section 7 an interactive protocol to re-randomize the polynomial. The third threat is that the model assumes that newly de-corrupted players lost all their memory, in particular their secret key. Thus the need of a protocol, denoted as *recover*, to give it again to them. Different techniques exist to recover lost shares. Following the seminal work of [Her+95] on proactive security, different protocols, e.g., [ZSR] [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. We propose another way to achieve recovery and to lower the communication cost, that leverages the redundant structure of our threshold encryption scheme, namely, our bivariate PVSS.

2 MODEL AND DEFINITIONS

2.1 Precise Model of Theorem 1

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 §7. 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 Arithmetic Circuits over $\mathbb{F}_p$, with input provision. Let us make precise the terminology "secure computation of an arithmetic circuit", 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 learn the final output. In each execution, The *robustness with input provision* guarantee is that, for any set of inputs $x_i \in \mathbb{F}_p^n$, if each player starts with input x_i, then all players receive the same output y, and y is a (random) evaluation of $F(x'_1, \ldots, x'_n)$ such that $x'_i = x_i$ for all indices i of uncorrupted players. The *privacy* guarantee is that the adversary learns no more than y, and even nothing if no recipient is corrupted.

2.1.2 The Almost Asynchronous Model, after [BHN10]. 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 broadcasted 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. Our results carry over the model where messages can be lost, provided a straightforward adaptation of the conditions for termination of protocols.

2.1.3 Transparent setup. We assume no trusted setup beyond the plain standard assumption, that [BCG20; TLP20] denote as a "bulletin board" of public keys. This can be formalized, e.g., as the functionality denoted certification authority $\mathcal{F}_{CA}$ in Canetti [Can04], to which each player can give at most one public key, and which outputs on demand the public key received from any given player. 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.

2.2 Cryptographic primitves

2.2.1 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 $f(0) = \sum_{i \in I} \lambda_i f(i)$.

2.2.2 Zero Knowledge (ZK) Proofs. A non-interactive zero-knowledge proof ($NIZKP$)[BFM88] allows a prover to convince a verifier that a statement is true without revealing any further information. A $NIZKP$ captures not only the truth of a statement but also that the prover "possesses" a witness w to this fact.

Let R be a relation and x a common input to the prover and the verifier. Following [CDK14, Definition 3 and 4], a non-interactive proof system Π for a relation R consists of two probabilistic polynomial algorithms: $\Pi.Prove$ and $\Pi.Verify$. More specifically, let f be a function and let's assume that the prover wants to demonstrate knowledge of a private witness w such that $f(w) = x$, or in other words, to prove the relation $R_f = \{x; w : f(w) = x\}$. The prover can produce a proof $\pi \leftarrow \Pi.Prove(x; w)$ while the verifier can invoke $\Pi.Verify(x, \pi)$ and either accepts or rejects the prover's claim. A $NIZKP$ scheme has two main properties. First, soundness requires that no prover can make the verifier accept a wrong statement except with some small probability. The upper bound of this probability is referred to as the soundness error of a proof system. For all non-uniform polynomial-time adversaries $\mathcal{A}$, we have:

$$Pr[\pi^* \leftarrow \mathcal{A}(x) : \Pi.Verify(x, \pi^*) = false] \approx 1 \tag{1}$$

Second, completeness ensures that for a valid prof, verification succeeds. For all $(x, w) \in R$, we have:

$$Pr[\pi \leftarrow \Pi.Prove(x, w) : \Pi.Verify(x, \pi) = true \text{ if } (x, w) \in R] = 1 \tag{2}$$

2.2.3 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 $\mathbb{F}_p$. 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. Let Π be a space denoted as "space of ZK proofs". Let $\mathsf{KeyGen} : \emptyset \rightarrow (s\mathcal{K}, p\mathcal{K})$ a PPT algorithm.

Let E be an efficiently computable PPT algorithm with source equal to $(p\mathcal{K}, \mathbb{F}_p)$, and which outputs one element in the union of the C_{pk}, appended with one element in Π. We will often abuse notations and also denote E for the first output only. More precisely, we have $E(\mathsf{pk} \in p\mathcal{K}, m \in \mathbb{F}_p) \in C_{\mathsf{pk}}$. 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 (sk, pk) is an output of KeyGen, and with target $\mathbb{F}_p \cup \{\mathsf{abort}\}$. We require the completeness condition, that $\mathsf{Dec}(\mathsf{sk}, E(\mathsf{pk}, m \in \mathbb{F}_p)) = m$. We require the classical IND-CPA *privacy* property, defined by a 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.

2.2.4 Example: Paillier. The Paillier encryption scheme, as e.g., recalled in [Ben+11, §2.1], has plaintext space $\mathbb{Z}/N\mathbb{Z}$ with $\mathsf{pk} := N$ a large product of two secret primes that 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 $[0, \ldots, p-1] \subset [0, \ldots, N-1]$, else it returns abort.

2.2.5 Example: El-Gamal 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 (G, g) be a group of prime order q, with public generator g, denoted multiplicatively, in which computing the DDH is hard. The

plaintext space of the baseline el Gamal encryption is the group G, which is isomorphic to $\mathbb{Z}/q\mathbb{Z}$. The ciphertext space is also G. Let us recall key generation, encryption and decryption. Let h be another public fixed random generator of G (for instance, obtained from a CRS). KeyGen is as follows. Samples $\mathsf{sk} \in \mathbb{Z}_q^*$ at random, and define $\mathsf{pk} := h^{\mathsf{sk}}$ as his public key. To encrypt $\gamma \in G$ under public key pk, sample $r \in \mathbb{F}_q$ at random and output $(\mathsf{pk}^r, h^r\gamma)$. Decryption is $\mathrm{Dec}\big(\mathsf{sk}, (ciph_1, ciph_2)\big) := ciph_2/(ciph_1)^{1/sk}$.

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 $[0, \ldots, p-1] \subset \mathbb{F}_q$. Then, in turn, we map $\mathbb{F}_q$ to G by $x \to g^x$, then apply the previously defined el-Gamal. This is where our terminology "in-the-exponent" comes from. Now, decryption of a ciphertext $c \in G$ consists in: applying the decryption of the baseline el Gamal to obtain some $g^x \in G$, 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 $x \in [0, \ldots, p-1]$ is found, then output $x \bmod p \in \mathbb{F}_p$. Else, output abort.

Thus, to make this work, we have another requirement on p to make here, which is that p is small enough such that every discrete log of absolute value smaller than p can be computed.

2.3 Reminder of [BHN10, PODC'10]

2.3.1 Protocol overview. In [BHN10], Beerliová-Trubíniová, Hirt and Nielsen proposed a MPC protocol that securely compute any arithmetic circuit over $\mathbb{F}_p$ as detailed in §2.1.1. Moreover it enforces *input provision* and tolerates $t < n/2$ corruptions in a almost asynchronous model. It follows the so-called *pre-processing model*. In this model, the protocol is split up into an offline (a.k.a. *pre-processing*) phase and an online phase. In the offline phase, the parties execute a protocol which emulates a trusted dealer who distributes correlated randomness to parties, that is then consumed in the online phase as the circuit is evaluated gate-by-gate. The offline phase is independent from both the inputs and the circuit and as such can be computed at any point prior the evaluation of the circuit. The correlated randomness produced by the offline phase, is so-called *multiplication triples*[Bea91].

In this specific setting, there are two main challenges: the asynchrony and the security threshold $t < n/2$. To implement asynchronous MPC, they use a threshold additive encryption scheme, whose definition is recalled in appendix §A. Unfortunately for $t < n/2$, byzantine agreement is not possible. As a result, agreement on the encrypted outputs of intermediary gates cannot be guaranteed. To solve this inconsistency of views, the whole circuit is evaluated many times in parallel, once for every player, denoted as *king*. The other players, acting as *slaves*, help the *king* to evaluate his copy of the circuit. This ensures that when the *king* is honest, all slaves have consistent views on all ciphertexts.

2.3.2 Protocol details. Let $\mathcal{E}$ denote a threshold additive homomorphic encryption, definition is recalled in appendix §A. Let F be the arithmetic circuit to be computed. For simplicity we consider a deterministic circuit here. How to evaluate random gates will be discussed and improved in §6. Starting with encryptions of the inputs (broadcasted in the initial round), the players jointly compute encryptions of the outputs of the intermediary gates, until eventually an encryption of the output is jointly decrypted (using threshold decryption). Addition gates are evaluated locally without interaction using the homomorphic properties of the encryption scheme while multiplication gates require interactions and the generation of multiplication triples [Bea91], defined as triples of two encrypted random values along with the encrypted product. More precisely, the protocol consists in the following steps:

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

(1) **Inputs broadcast**: Each party P_i broadcast its encrypted input $\mathcal{E}_{\mathsf{pk}}(x_i)$. 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**: To generate an encrypted multiplication triple unknown to the adversary $\mathcal{A}$, 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, which guarantees that a chain of $t+1$ consecutive randomizations is achieved. Thus the plaintext values of the factors of the encrypted triple, are indistinguishable to the adversary from uniform random ones.
(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. and outputs $\mathcal{E}_{\mathsf{pk}}(F(x_1, x_2, ..., x_n))$
(4) **Termination**: Each circuit output is jointly decrypted and the king learns the result z. Then it sends a signature request on z to all parties and continues to act as slaves. When it received signature shares from $t+1$ parties, it broadcasts the signed output. Once $t+1$ kings have finished with the *same signed output*, then necessarily this must be the correct one and all players adopt it.

3 METHOD OVERVIEW

3.1 Computation structure

We gradually present the structure of computation to evaluate a circuit as introduced in §1.1.5.

3.1.1 Stage. 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 aggregation of $t+1$ signatures on the result. Given a ***Verif0ut***, we use ***Verif0ut***.value and ***Verif0ut***.QVC to refer to the above-mentioned elements. Throughout the computation, we maintain the following invariant from the distribution to the termination:

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

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 [BHN10], this requires every player to wait until they receive $t+1$ identical results to be sure that at least one honest king learns the correct result. In our protocol a signed value is correct (per Inv_stage). Upon receiving one correct output a player multicasts it and immediately terminates.

Examples. Some of the basic stage functions used throughout the protocol are defined as particular Stages. The following basic functions will be used to compute any circuit: *i*) TAE.**PubDec** that enables to publicly decrypt an output *ii*) TAE.**Rand** that generates a verifiable random variable and presented in 6.3 *iii*) TAE.**Add** that enables to add two outputs together and introduced in 3.1.5 *iv*) TAE.**Mult** that enables to multiply an output with a scalar as shown in 4.3.

3.1.2 Overall structure of a Stage. A king drives a Stage in just two exchanges of messages called *phases*. The first phase (referred as *contribution phase)* guarantees that each player is convinced of the correctness of the computation made

during the current stage. The king is responsible for the collection of partial contributions, and for the aggregation of the individual Zero-Knowledge Proofs (ZKP) into a ***Combine Proof*** (CP is short) that validates all computations made during a stage. The second phase (referred as *verification phase*) guarantees the correctness of a value up to the current step through the formation of a ***Quorum Verification Certificate***[1] (QVC) consisting of $t+1$ signatures. At the end of a stage, all outputs must be associated with a Quorum Verification Certificate. Thus, a stage is defined by three elements:

- $contrib_{sid}$ [2] a private contribution function for stage SID invoked by all slaves
- s_j: a secret material used by P_j to compute $contrib_{sid}$
- $combine_{sid}$ a public aggregation function used by the king to aggregate the partial contributions received from the slaves.

In summary, a stage takes as inputs a set of ***verified stage outputs*** $\{X_i\}_i$ and produce another ***VerifOut*** that is the result of the function $combine_{sid}$ applied on $t+1$ contributions $contrib_{sid}(\{X_i\}_i)$ from a set $\mathcal{S}$ of parties . In other words, ***Verif0ut***.value = $combine_{sid}(\{contrib_{sid}(\{X_i\}_i, s_j\}_{j\in\mathcal{S},|\mathcal{S}|=t+1})$. The execution of a Stage for a player is presented in figure 1, and a more complete description of the data structures used and the pseudocodes are given in the appendix B.

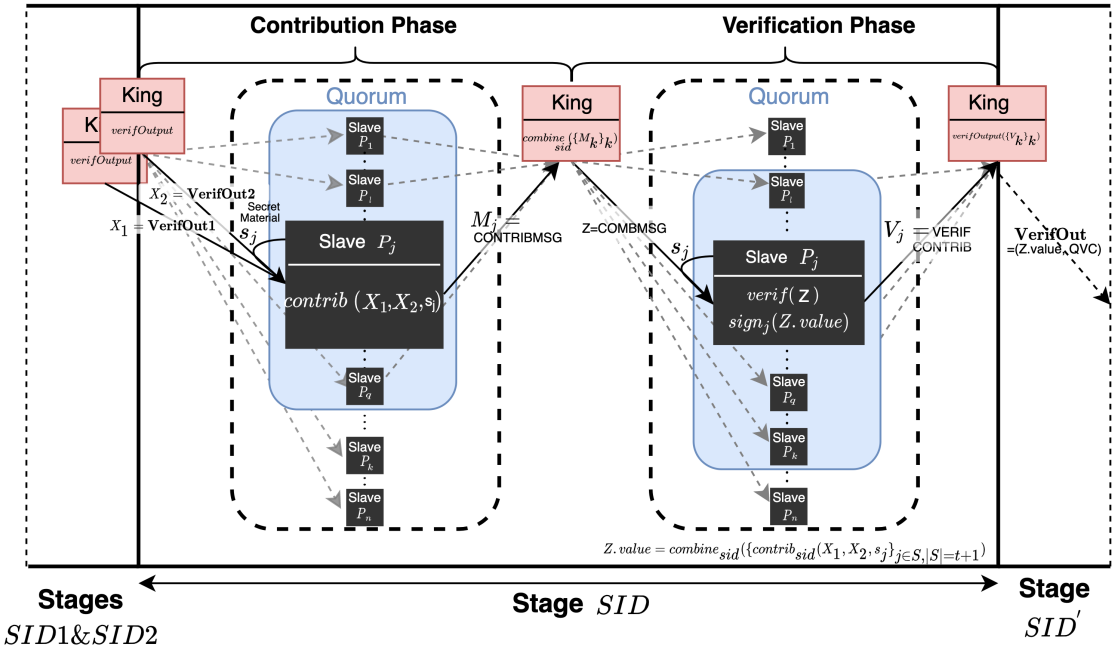

Fig. 1. Computation stage for a player P_j. It first receives two ***verified stage outputs*** X_1 and X_2 from stages SID_1 and SID_2 and uses its secret material s_j to compute its partial contribution using $contrib_{sid}$. The king collects $t+1$ $CONTRIBMSG$ messages with valid proofs, combines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

[1] A *quorum* denotes a subset of players in $\mathcal{P}$ of size at least $t+1$
[2] To simplify notation, here sid denotes $SID.function$

3.1.3 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*** and multicasts the result in a $COMBMSG$ message. Recall that any player can verify a proof using the function $\Pi_{sid}.Verify$.

3.1.4 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***. 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 $SigVerify$.

3.1.5 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 4.3, 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 to 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.

4 THRESHOLD-ADDITIVE ENCRYPTION (TAE) WITH TRANSPARENT SETUP

We gradually define a *Threshold-Additive Encryption* (TAE) scheme with transparent setup, with plaintext space a finite field, following the programme presented in the introduction §1.1.4.

Basic specifications. In §4.1 we, firstly, require a TAE to be a verifiable 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 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.

Advanced specifications. are then specified in §4.2. They all come as triples of algorithms, with exactly the same structure as $(t+1)$-threshold description. Apart from addition and multiplication by a scalar, we also specify a $(t+1)$-threshold "private decryption": **PrivDec**, which outputs the plaintext only to a designated recipient. Regarding this latter, on the one hand, the implementation will be simple from a conceptual perspective: each player applies Contrib on the ciphertext, then encrypts the output under the recipient's public key. However, enabling public verifiability of this operation will require more complicated ZK proofs.

In §4.3 we provide an implementation of TAE from any public key encryption scheme in the sense of 2.2.3. In detail, a ciphertext is a $n \times n$ sized matrix of ciphertexts of evaluations of a bivariate symmetric polynomial, with possibly empty entries. In §4.4 we wrap TAE in our computation framework of interactive stages, in order to apply it to our problem of asynchronous MPC without trusted setup. In §4.6 we finally instantiate the ZK proofs needed for implementing TAE from two possible baseline encryption schemes: both el Gamal in-the-exponent, and Paillier.

4.1 Basic specification: a verifiable threshold encryption scheme with transparent setup

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 el Gamal scheme as baseline (anticipating on §4.3), then p is small enough so this baseline (cf §2.2.3) has efficient decryption.

Definition 3. Let $\mathbb{F}_p$ be a finite field. A *verifiable* $(t+1)$*-out-of* n *threshold encryption scheme with transparent setup* over $\mathbb{F}_p$ is the data of a space $\boldsymbol{\mathcal{C}}$ denoted as the *global ciphertext space*, spaces $s\mathcal{K}$ and $p\mathcal{K}$ denoted as the "secret keys" and "public keys" spaces, a space Π denoted the "space of ZK proofs", of two PPT algorithms

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

Encrypt: $p\mathcal{K}^n \times \mathbb{F}_p \to \boldsymbol{\mathcal{C}} \times \Pi$ denoted "encryption algorithm", and of four deterministic algorithms:

Encrypt. Verify: $p\mathcal{K}^n \times \boldsymbol{\mathcal{C}} \times \Pi \to \{\text{accept}, \text{reject}\}$,

PubDec. Contrib: $s\mathcal{K} \times \boldsymbol{\mathcal{C}} \to \left\{\{0,1\}^* \times \Pi\right\} \cup$ abort, denoted as "decryption share". The $\{0,1\}^*$ denotes binary strings of unspecified lengths, but in our implementation it will be a vector of n elements of the plaintext space.

PubDec. Verify: $p\mathcal{K} \times \boldsymbol{\mathcal{C}} \times \{0,1\}^* \times \Pi \to \{\text{accept}, \text{reject}\}$ which proves correctness of a decryption share, and

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

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

Completeness. For any $\mathbf{pk} \in p\mathcal{K}^n$, $m \in \mathbb{F}_p$, and $(c, \pi) :=$ **Encrypt**(pk, m), then **Encrypt**.Verify(pk, c, π) = accept. For any $\mathbf{pk} \in p\mathcal{K}^n$, suppose that there is a subset $\mathcal{I} \subset \{1, \ldots, n\}$ of $t+1$ indices, such that, for all $i \in \mathcal{I}$, we have that $\mathsf{pk}_i := \mathbf{pk}[i]$ is the public key of a correctly (locally) generated key pair: $(\mathsf{sk}_i, \mathsf{pk}_i) = KeyGen()$. Then for all $m \in \mathbb{F}_p$, denote $(c, \pi) :=$ **Encrypt**($\mathbf{pk}, m$) and $(d_i^c, \pi_i) :=$ **PubDec**.Contrib(sk_i, c) $\forall i \in \mathcal{I}$, we have both that **PubDec**.Verify($\mathsf{pk}_i, c, d_i^c, \pi_i$) = accept $\forall i \in \mathcal{I}$ and $m =$ **PubDec**.Combine$\left((d_i^c)_{i \in \mathcal{I}}\right)$.

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. *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 negligible advantage in distinguishing whether c is an encryption of m_0 or m_1.

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 undistinguishable from valid decryption shares from the remaining player indices, even for an adversary holding the secret keys of the corrupt indices.

Decryption consistency. Is defined as in appendix §A. Namely, on input a complete set of n correctly generated key pairs, then adversary cannot forge, except with negligible probability, a ($c \in \boldsymbol{\mathcal{C}}, \pi \in \Pi$) which would be accepted by **Encrypt**.Verify, along with two sets of $t+1$ strings accepted by **PubDec**.Verify as being valid outputs of **PubDec**.Contrib, and such that their **PubDec**.Combine are different.

4.1.1 Remark: Implementation with a PVSS.. Notice that a scheme with the basic requirements above can be implemented from any verifiable public key encryption scheme E, with the conditions and conventions of §2.2.3, as follows. Encryption

of a plaintext $m \in \mathbb{F}_p$ consists in: sampling a degree $t+1$ polynomial $B \in \mathbb{F}_p[X]$ such that $B(0) = m$ uniformly at random; computing its evaluations $B(\alpha_i)$; output of the vector of encryptions of these evaluations under the public keys of the corresponding players: $\big[E(\mathsf{pk}_i, B(\alpha_i)),\ i \in [n]\big]$. The second output of **Encrypt** is a ZK proof of correctness of the computation of this vector (with witness the plaintext). $(t+1)$-threshold decryption proceeds by decryption of any $t+1$ coordinates, then interpolation of B to recover $B(0) = m$. This is exactly [GMW91, §3.3], later rediscovered as "PVSS".

4.2 Advanced specifications: private opening, and interactive homomorphic operations

We fix E any public key encryption scheme as in §2.2.3. We abuse notations and also denote as pk_i the public keys used for E . This abuse is because, in our implementation §4.3, E will be the baseline public key encryption scheme, thus the public keys will coincide.

Definition 4 (TAE)**.** A $(t+1)$-out-of n TAE over $\mathbb{F}_p$, is the data of a verifiable threshold encryption scheme with transparent setup, as defined in Definition 3, of which we keep the notations, along with the following PPT algorithms. Notice that **Add** is just a particular case of **LinComb**, which we describe for clarity:

PrivDec. Contrib $: p\mathcal{K} \times s\mathcal{K} \times \mathcal{C} \to \{\{0,1\}^* \times \Pi\} \cup$ abort. On input pk_r, which is the recipient's public key, sk_i and c, outputs $E(\mathsf{pk}_r,$ **PubDec**.Contrib$(sk_i, c))$ and $\pi \in \Pi\big)$, 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 $\mathbb{F}_p$

PrivDec. Verify $: p\mathcal{K} \times p\mathcal{K} \times \{0,1\}^* \times \Pi \to \{$accept, reject$\}$ proves correctness of the computation of the $PrivDec$.Contrib.

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

Add. Verify with parameter a player index: $p\mathcal{K}^n \times \mathcal{C}^2 \times \{0,1\}^* \times \Pi \to \{$accept, reject$\}$

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 $\big(\lambda_1, \ldots, \lambda_L\big) \in \mathbb{F}_p^L : p\mathcal{K}^n \times s\mathcal{K} \times \mathcal{C}^L \to \{\{0,1\}^* \times \Pi\} \cup$ abort.

LinComb. Verify $: p\mathcal{K}^n \times \mathcal{C}^L \times \{0,1\}^* \times \Pi \to \{$accept, reject$\}$ proves correctness of the computation of a Contrib.

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 (updated below) & simulatability of decryption shares (unchanged), and decryption consistency* as defined below.

Completeness. To define completeness, we firstly introduce the following recursive definition.

Definition 5 (TAE.ciphertext)**.** First, any correctly computed **Encrypt**(x) any $x \in \mathbb{F}_p$ 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 combinaison 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. Finally, the completeness guarantees that the proofs output by any correctly computed Contrib, is accepted by the corresponding Verify.

IND-CPA.. We consider the same game as in §4.1. In addition we allow the adversary, *before and after* he receives the encryption of one of his challenge plaintexts, 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 **PrivDec**.Contrib, the recipient is an *uncorrupt* player. Indeed, otherwise, this would enable to adversary to obtain the decryption of any ciphertext, i.e. as in IND-CCA, which we do not guarantee. Then, apart from this modification, the privacy requirement is unchanged.

Decryption consistency. We require that the adversary cannot produce any TAE.ciphertext along with two sets of $t+1$ **PubDec** shares, that would both be all accepted as valid, and such that their **PubDec**.Combine would return two different outputs.

Let us make a remark on this definition, anticipating on our implementation. Notice that, by soundness of ZK proofs, any $c \in \mathcal{C}$ which is accepted as **Encrypt**.Verify is in particular a TAE.ciphertext. Thus, the previous requirement is a generalization of decryption consistency as defined in §4.1.

4.2.1 A comment on the power of the adversary in the IND-CPA game. Recall that we allow the adversary to query contributions of **PrivDec**, and **LinComb**. Actually, the adversary and the simulator have no more power than in the definition of threshold homomorphic encryption in [CDN01] (see also Definition 13 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.

4.3 Implementing TAE

We use the notations of §2.2. We consider a public key encryption scheme (KeyGen, E, Dec) over $\mathbb{F}_p$ as defined in §2.2.3, 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 7, with respect to the following implementation of TAE, this definition will turn out to be synonymous Definition 5 of a TAE.ciphertext.

Definition 6. 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:

$$c_{i,j} := E_j(B(\alpha_i, \alpha_j)) \,, \forall i \in \mathcal{I} \,, j \in [n] \tag{4}$$

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.

4.3.1 ***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: $c_{m,(ij)} := E_j(B(\alpha_i, \alpha_j))$, $\forall i \in \mathcal{I}$, $j \in [n]$, and also output a ZK proof $\pi_{Encrypt}$ that proves correctness of the computation of the output, namely, the relation $R_{Encrypt}$ as presented in appendix C.

4.3.2 Threshold decryption. **PubDec.**Contrib: Let (sk_j, c) be the inputs. By Definition 6, 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. Each P_j then outputs $(d^c_{i,j})_{i \in \mathcal{I}_j}$, along with a proof that $c_{i,j} \in E(\mathsf{pk}_j, d^c_{i,j})$ $\forall i \in \mathcal{I}_j$.

PubDec.Combine: on input $t + 1$ decryption shares (with proofs returned as accept), 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 6, 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: $[E_r(d^c_{i,j})$, $i \in \mathcal{I}]$ The ZK proof output by **PrivDec**.Contrib, $\pi_{PrivDec_r,j}$, proves correctness of the output, namely: the relation $R_{PrivDec_r,j}$ presented in appendix C.

PrivDec.Combine Initialize an empty $n \times n$ array. For each correctly checked contribution $[c^{(out)}_{i,j}$, $i \in \mathcal{I}]$, from P_j, copy the elements of this list at their positions (i, j) in the array. After receiving $t + 1$ contributions with correct proofs, output the array. The combine proof $\pi_{\textbf{PrivDec}}$ is the trivial concatenation of the correct proofs received from the $t + 1$ players.

4.3.3 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 6, 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: $\left(d^c_{i,j}\right)_{i \in \mathcal{I}}$ and $\left(d^{c'}_{i,j}\right)_{i \in \mathcal{I}'}$. 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 $[d^{c+c'}_{i,j}$, $i \in [n]]$. Finally, output encryptions of its entries, into the form of a n-sized row vector, namely:

$$c^{(out)}_j := \left[E_i(d^{c+c'}_{j,i}) \, , i \in [n]\right] . \tag{5}$$

The ZK proof $\pi_{\textbf{Add},j}$ output proves that these computations were done correctly, namely, proves the relation $R_{\textbf{Add}}$ presented in appendix C

Add.Combine: Initiate an empty $n \times n$ array, that is, filled with $\perp$. For each contribution $c^{(out)}_j$, 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 correctly checked addition shares with correct proofs, output the array computed so far.

4.3.4 *Proof of completeness, privacy: IND-CPA & shares simulatability, and decryption consistency.*

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 6, 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 7. *With respect to the implementations of **Encrypt**, **Add** and more generally **LinComb** above, we have that the property of being a* TAE.ciphertext *(Definition 5) 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 6 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 5, 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 a 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 itself 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 8.

Lemma 8. *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 8. □

Privacy: shares simulatability. By proposition 7, 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 7, 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 guarantee 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 $B(0)$.

4.4 Wrapping TAE in our computation stages framework, for application to MPC

We now compile the previous specification of a TAE, of Definition 4, 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 $\boldsymbol{\mathcal{C}}$ 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*** $p\mathcal{K}^n \times \boldsymbol{\mathcal{C}}^* \times \Pi^* \to \{(\boldsymbol{\mathcal{C}} \times \Pi)^n\}$ 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 $c_{m_i} = \textbf{Encrypt}(\mathsf{pk}, m_i)$ 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 $c_j := \textbf{Encrypt}(\mathsf{pk}, 0)$ in the vector box j, and adds a proof of correct encryption.
- TAE.***PubDec*** $s\mathcal{K}^n \times \boldsymbol{\mathcal{C}} \to \mathbb{F}_p$ is a stage that takes as input a ***verified*** TAE.ciphertext c and produces a verified plaintext m such that $m \leftarrow \mathsf{TAE}.\textbf{PubDec}(\textbf{Encrypt}(\mathsf{pk}, m))$.

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 $m = \mathsf{TAE}.\textbf{PubDec}(c)$. 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*** $s\mathcal{K}^n \times p\mathcal{K} \times \boldsymbol{\mathcal{C}} \rightarrow C^*$ is a stage that takes as input a verified TAE.ciphertext c_m and a designated player 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*** $p\mathcal{K}^n \times s\mathcal{K}^n \times \boldsymbol{\mathcal{C}} \times \boldsymbol{\mathcal{C}} \rightarrow \boldsymbol{\mathcal{C}}$ 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 $m + m'$.

4.5 (Informal) Proof of theorem 1

We sketch how the properties argued in Theorem 1 derive from [BHN10, Theorem 1] reminded in 2.3 and our TAE introduced in Section 4. We first note that our protocol encompasses the one of [BHN10]. and mainly differs by replacing their *AHE* with our TAE stages defined in Section 4.4. In detail, considering a TAE with the notations introduced in §4 and using the structure introduced in §2.3.2: (0) we replace the trusted setup by a local invocation to **Keygen**(). Then all players publish their public keys on the bulletin board. Step (1) is subdivided in our model into two sub-steps. First each player broadcasts their inputs, encrypted with **Encrypt**, then they invoke TAE.**Input** on them. (2) and (3) are unchanged, except that homomorphic linear combinations, which were computed locally on $\mathcal{E}-$ciphertexts, are replaced by TAE.**LinComb** applied to TAE.ciphertexts. (4) The final gate outputs the **PubDec** of $F(x_1, .., x_n)$. If not all players are recipients, then the straightforward modification is done by using **PrivDec** instead. When a player receives these ***verified stage outputs*** from one king then he forwards them to all players and halts. Given these preconditions, it is easy to see that Theorem 1 holds.

Remark 9. Rerandomization of ciphertexts, as used in [BHN10] in (2) to generate multiplication triples, is not needed anymore, since our implementation of **Add** and **Mult** involves reencryption of the well formed ciphertexts.

Remark 10. In (4), 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 [BHN10], he needs to wait to receive identical signed plaintext outputs from $t + 1$ kings before halting.

4.6 Instantiations of the ZK proofs needed

Since any NP language is provable in zero-knowledge ([GMW91]), it follows from our implementation above §4.3 that a TAE can be instantiated from any public key encryption scheme. However not all public key encryption schemes come with natural ZK-proof systems. Using generic ZK-proof systems may not be practical. In what follows we sketch how to instantiate our program with the Paillier and el Gamal schemes as defined in §2.2.3. In both cases, our baseline for the ZK proofs is the recent framework of Attema-Cramer [AC20].

4.6.1 From el Gamal in-the-exponent scheme. The invariant of the proof scheme [AC20] is that it enables any prover, which exhibits one (or several) Pedersen commitment(s) to one (or several) secret input x, which we denote $Com(x)$, to prove that the opening of this commitment satisfies any public relation R expressed by an arithmetic circuit in $\mathbb{F}_q$. Plus, it turns out that the second part of an el Gamal in-the-exponent encryption of x: $(h^\gamma g^x)$, is none other than a Pedersen commitment to x with hiding parameter γ. Overall, this enables to carry out the ZK proofs required in 4.3 in a modular way. For instance, to prove Relation $R_{\textbf{Add}}$, the prover sends a unique commitment to the witness, including the various

d_{ij} (with the notations of §C), along with separate proofs that these committed witnesses satisfy the various conditions needed. (For instance, proving that an el Gamal ciphertext encrypts a committed value is used in [Sch99] with the proof denoted as "Chaum-Pedersen" for equality of discrete logs) This now appears as a subcase of the framework of [AC20]. See also [AC20, §7] for proofs that committed values lie in a certain range (in our case: $[0, \ldots, p-1]$), details are given in the full version of [AC20].

4.6.2 From Pailler scheme. The starting point is the ZK proof in [DJ02, Appendix A] that proves that a Paillier ciphertexts encrypts a Damgård-Fujisaki commitment [DF02]. From this point, all ZK proofs can be carried over Damgård-Fujisaki commitments, thanks to [AC20, §8], which re-build their framework over this commitment (instead of Pedersen's).

5 MULTIPLICATION TRIPLES GENERATION IN THE ALMOST ASYNCHRONOUS MODEL FROM PREPROCESSING

We present our protocol to produce multiplication triples. In §5.1, we detail a roadmap to prove Theorem 2. In 5.2, we describe the main building blocks, and finally in §5.3, we summarize how it all comes together.

5.1 Method overview

In order to multiply secrets, a mainstream approach, since Beaver [Bea91], consists in having players precompute random secret multiplication triples as defined in section 2.3, that are later used to evaluate a circuit. In [CHP13], Choudhury, Hirt and Patra have developed a new approach to produce multiplication triples: first, players share random multiplication triples, and from that, shared random triples unknown to the adversary $\mathcal{A}$ are extracted. We follow this approach and augment it by adding verifiability thanks to Zero-Knowledge proofs. This allows us to make structural modifications to the protocol which have the result of increasing the number of triples generated and to improve resiliency from $t < n/4$ to $t < n/2$, *without* consensus.

Our framework is made up of three main phases:

(1) **Triple sharing** We first allow every player P_i to share multiplication triples to all other players. The first round of synchronous broadcast and our encryption method ensure *input provision* and *sharing correctness*. Moreover, for an honest P_i, the shared triples remain private from $\mathcal{A}$.
(2) **Verifiable transformation of independent multiplication triples to co-related triples** The second module allows every player P_i to verify that the shared triples are indeed multiplication triples. Unlike [CHP13], the verification is local and deterministic. This ensures that all honest players own the same correct multiplication without the need of a Byzantine Agreement, which considerably simplifies the protocol.
(3) **Randomness extraction from co-related triples** Finally, the third module allows every honest player to extract random multiplication triples unknown to $\mathcal{A}$ from the shared material.

This method is independent of our the threshold additive encryption scheme. It can be instantiated either with the one considered in [CDN01; BHN10], which requires trusted setup, of ours in §4, which does not. Thus, we adopt generic notations: $\mathcal{E}$ denotes any threshold encryption function that enables (possibly with interactions as for the TAE scheme presented in Section 4) the addition of ciphertexts (noted $\boxplus$) and the scalar multiplication (noted $\boxdot$).

5.2 Main building blocks

Non-interactive Proof of plaintext multiplication: We present 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)). We introduce such a verification function $MultVerif$ that can be constructed from [DJ01, §4.2] and presented in figure 6.

Transformation of independent multiplication triples to co-related triples: The idea is to interpolate three polynomials $\mathbb{x}(.), \mathbb{y}(.)$ and $\mathbb{z}(.)$ from the shared triples and use them to produce new co-related shared 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. The deterministic aspect is essential in our computation method in order to ensure the correctness of the computation.

In greater detail, protocol $TripTrans$ takes as input $t + 1 + t'$ *correct* and *independent* shared 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 $c^{(j)} = a^{(j)}.b^{(j)}$. It outputs $t + 1 + t'$ "co-related" shared 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 [BFO12]. Specifically, the two polynomials $\mathbb{x}$ and $\mathbb{y}$ are entirely defined by the first and second components $(a^{(i)}, b^{(i)})$ of the first $\frac{t+t'}{2} + 1$ triples. The construction of $\mathbb{z}(.)$ is not as straightforward due to the difference in degree. We use $\mathbb{x}(.)$ and $\mathbb{y}(.)$ to compute $\frac{t+t'}{2}$ "new points" and use the remaining $\frac{t+t'}{2}$ available triples $(A^{(i)}, B^{(i)}, C^{(i)})_{i\in[\frac{t+t'}{2}+2, t+1+t']}$ to compute their products . Ultimately, $\mathbb{z}(.)$ is both defined by the last components of the first $\frac{t+t'}{2} + 1$ triples and by the $\frac{t+t'}{2}$ computed products. Details are presented in figure 7.

Randomness extraction: We present a protocol called $TripExt$ that extract a random multiplication triples unknown to $\mathcal{A}$ from a set of $(t + 1 + t')$ shared multiplication triples. The idea of the protocol is inherited from [CHP13] and can be summed up as follows: from a correct multiplication triples shared by the $t + 1 + t'$ players the transformation protocol is executed on 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 output multiplication triple, unknown to $\mathcal{A}$, is then extracted as $\{(\mathbb{X}(\beta), \mathbb{Y}(\beta), \mathbb{Z}(\beta))\}$. Details are presented in figure 9.

5.3 The Preprocessing phase protocol

Our preprocessing phase protocol now consists of the following steps: **(1)** Every player P_i acts as a dealer and shares a random multiplication triple using an instance of *Share*. **(2)** Each player then verifies the correctness of the multiplication triples and outputs a common set $\mathcal{U}$ of $t+1+t'$ dealers who have correctly shared multiplication triple in their respective sharing instances. **(3)** Finally, each player executes the triple extraction protocol $TripExt$ on the set of triples shared by players in $\mathcal{U}$ to extract a random multiplication triples unknown to $\mathcal{A}$. The protocol is given in figure 10.

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

5.4 (Informal) Proof of theorem 2

We first note that properties of the preprocessing protocol *PreProc* presented in section D.3 imply that all the honest players will terminate the protocol and will output a random multiplication triples unknown to the adversary.

We briefly recall from Section §3.1 and §4 that every homomorphic linear combination (in particular $\boxplus$ and $\boxdot$), possibly followed by a public decryption, is computed in an overall four consecutive interactions, which we denoted a stage. In particular, following the protocol detailed in this section, the generation of a multiplication triple requires a constant number of consecutive stages, which is independent of the number of players.

6 ON-THE-FLY ENCRYPTED RANDOM VALUE GENERATION

We propose a linear threshold construction to produce an encrypted random value without setup. This construction makes possible the generation of pairs of public/private keys as well as proactive security. Let 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 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}.

6.1 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.

6.1.1 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.

6.2 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.**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

6.3 On-the-fly implementation of TAE.Rand () without broadcast

To implement 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.

6.3.1 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 initialisation 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

$$PRSS(l, a) := \sum_{|A|=n-t,\ l\in A} \psi_{r_A}(a) \cdot f_A(l) \tag{6}$$

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

6.3.2 Our additional ingredients. We enrich the *PRSS* 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 $\mathcal{C}$, 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, let us illustrate an implementation of the previous ingredients, based on the one of our TAE in §4.3. 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.

6.3.3 Our on the fly threshold-encrypted random generation. The initial step is to implement the trusted dealer of PRSS keys, by the distributed key generation protocol of §6.2. 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 §6.3.2, 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 6.3.2. Its combination function simply takes as input a set of contributions issued by any set $\mathcal{L} \subset \{1,\dots,n\}$ of $t+1$ distinct players: $\{(c^l, \pi^l), l \in \mathcal{L}\}$, 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

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

Proposition 11. *The output of these two consecutive stages has the unpredictability property defined as in the game below.*

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 $1/p$.

6.3.4 Proof of Proposition 11.

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: **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: $\widetilde{PRSS(j,a)}_{j\in[n]\setminus\mathcal{J}}$, which he forwards to $\mathcal{A}$ as the challenge. Recall that, by construction, if c_b is a TAE.ciphertext of $s(a)$, then these prospective

uncorrupt encryptions are exactly TAE.**Rand** contributions of uncorrupt players indices. Therefore, if we are in this case, then $\mathcal{A}$ has nonnegligible distinguishing advantage.

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

7 PROACTIVE SECURITY

In §2.1 we define the model, denoted as "proactive". We then explain how it stands between the models of Liskov-Liskov-Schultz [SLL10], and of Baron-El Defrawy-Lampkins-Ostrovsky [Bar+14]. We finally compare it to the one of Cachin-Kursawe-Lysyanskaya-Strobl [Cac+02]. We then address the three threats mentionned in the introduction. Namely, we describe in §7.1.1 how to refresh the keys, both for encryption and for the randomness generation (§6), then in §7.2 how to refresh the plaintext shares constituting the ciphertexts.

7.1 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 §7.1.1 and §7.2, 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, players regularly erase from their memory all the material not needed anymore. Let us outline the chronology of a closing

7.1.1 Closing of an epoch. First, players in epoch e perform a protocol to generate a new public / private key pairs for all of them relatively to epoch $e + 1$. On the one hand, freshly decorrupted players have had their memory erased. Thus they locally generate a new public / private key pair, and publish the public key on the bulletin board. On the other hand, for the players who still have their keys, there are two alternatives. The simple one is to do the same, namely, each of them publishes a new 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. The more complicated one, which has the advantage not to use the bulletin board, is to perform the Keygen protocol of §6.2, once for each recipient player who still has its keys. This creates, for each player j, a new key pair, such that: the private key comes as TAE.ciphertexts, with the signature of $t + 1$ players attesting its correctness, which is furthermore privately opened to j, and, such that the public key is publicly opened.

Next, players generate new PRSS keys. For this they perform $\binom{n}{n-t}$ executions of the Keygen protocol of §6.2. Each execution has parameter a set A of $n - t$ recipient players.

Finally, players refresh the ciphertexts which are to be used in future epochs, as sketched in §1.2.3 and detailed below in §7.2. Then they delete their secret keys of previous epochs, and all plaintexts and ciphertexts related to previous epochs.

7.1.2 Similarities with [Bar+14]. The model of [Bar+14], is defined under a synchrony assumption where the time is divided in 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 $e + 1$.

7.1.3 Differences [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 to assume anyway, like we did in §7.1.1, that freshly decorrupted players have access to a public bulletin board of keys at the beginning of each epoch.

This allows us not to make the forward-security assumption. The advantage of not making this assumption, is that we have access to the encryption schemes of Paillier and el-Gamal-in-the-exponent. Hence, they enable efficient ZK proof systems, as required by our implementation of 4.3, of whom we sketch an efficient instantiation in 4.6.

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, 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 §7.2, takes a (small) constant number of stages.

7.1.4 Differences with Cachin-Kursawe-Lysyanskaya-Strobl [Cac+02]. 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[4]. 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.

[4] "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."

7.2 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. 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.**Add** to compute the summation of these $t+1$ contributions, which is denoted c_0, along with c_s. We prove with lemma 12 the privacy of this refresh.

Lemma 12. *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.*

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 §4.3.4 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 8 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. □

REFERENCES

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

[Bac+14] Michael Backes et al. "Asynchronous MPC with a Strict Honest Majority Using Non-equivocation". In: *ACM Symposium on Principles of Distributed Computing 2014*. ACM, July 2014, pp. 10–19.

[Bar+14] Joshua Baron et al. "How to Withstand Mobile Virus Attacks, Revisited". In: PODC '14. 2014.

[BCG20] Elette Boyle, Ran Cohen, and Aarushi Goel. "Breaking the $O(\sqrt{n})$-Bits Barrier: Balanced Byzantine Agreement with Polylog Bits Per-Party". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 130.

[Bea91] Donald Beaver. "Foundations of Secure Interactive Computing". In: *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '91. Berlin, Heidelberg: Springer-Verlag, 1991, pp. 377–391. ISBN: 3540551883.

[Ben+11] Rikke Bendlin et al. "Semi-homomorphic Encryption and Multiparty Computation". In: *Advances in Cryptology EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. 2011.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-Interactive Zero-Knowledge and Its Applications". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112. ISBN: 0897912640. DOI: 10.1145/62212.62222. URL: https://doi.org/10.1145/62212.62222.

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

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. "Perfectly-Secure MPC with Linear Communication Complexity". In: *Theory of Cryptography*. 2008.

[BHN10] Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. "On the theoretical gap between synchronous and asynchronous MPC protocols". In: *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. 2010.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. "Asynchronous Secure Computations with Optimal Resilience (Extended Abstract)". In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '94. Los Angeles, California, USA: Association for Computing Machinery, 1994, pp. 183–192. ISBN: 0897916549. DOI: 10.1145/197917.198088. URL: https://doi.org/10.1145/197917.198088.

[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, Sydney, Australia, November 9-11, 1999, Proceedings*. Ed. by Vijay Varadharajan and Yi Mu. 1999.

[Cac+02] Christian Cachin et al. "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. Washington, DC, USA: Association for Computing Machinery, 2002, pp. 88–97. ISBN: 1581136129. DOI: 10.1145/586110.586124. URL: https://doi.org/10.1145/586110.586124.

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

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

[CDI05] Ronald Cramer, Ivan Damgaard, and Yuval Ishai. "Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation". In: *Proceedings of the Second International Conference on Theory of Cryptography*. 2005.

[CDK14] Ronald Cramer, Ivan Damgård, and Marcel Keller. "On the Amortized Complexity of Zero-Knowledge Protocols". English. In: *Journal of Cryptology* 27.2 (2014), pp. 284–316. ISSN: 0933-2790. DOI: 10.1007/s00145-013-9145-x.

[CDN01] Ronald Cramer, Ivan Damgaard, and Jesper B. Nielsen. "Multiparty Computation from Threshold Homomorphic Encryption". In: *Advances in Cryptology — EUROCRYPT 2001*. Springer Berlin Heidelberg, 2001.

[CFY16] R. K. Cunningham, Benjamin Fuller, and Sophia Yakoubov. "Catching MPC Cheaters: Identification and Openability". In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 611.

[Cho+13] Ashish Choudhury et al. *Between a Rock and a Hard Place: Interpolating Between MPC and FHE*. Cryptology ePrint Archive, Report 2013/085. https://eprint.iacr.org/2013/085. 2013.

[CHP13] Ashish Choudhury, Martin Hirt, and Arpita Patra. "Asynchronous Multiparty Computation with Linear Communication Complexity". In: *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*. DISC 2013. Jerusalem, Israel: Springer-Verlag, 2013, pp. 388–402.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography". In: *J. Cryptol.* 18.3 (July 2005), pp. 219–246. ISSN: 0933-2790. DOI: 10.1007/s00145-005-0318-0. URL: https://doi.org/10.1007/s00145-005-0318-0.

[CS03] Jan Camenisch and Victor Shoup. "Practical verifiable encryption and decryption of discrete logarithms". English (US). In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Dan Boneh. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Copyright: Copyright 2020 Elsevier B.V., All rights reserved. Springer Verlag, 2003, pp. 126–144. ISBN: 9783540406747. DOI: 10.1007/978-3-540-45146-4_8.

[Dam+09] I. Damgård et al. "Asynchronous multiparty computation: Theory and Implementation". In: *Proceedings of the 12th International Conference on Practice and Theory in Public-Key Cryptography Public Key Cryptography (PKC 2009), 18-20 March 2009, Irvine, CA, USA*. Ed. by S. Jarecki and G. Tsudik. Springer, 2009.

[Daz+08] V. Daza et al. "Ad-Hoc Threshold Broadcast Encryption with Shorter Ciphertexts". In: *Electron. Notes Theor. Comput. Sci.* 192 (2008), pp. 3–15.

[DF02] Ivan Damgård and Eiichiro Fujisaki. "A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order". In: *Advances in Cryptology — ASIACRYPT 2002*. 2002.

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

[DJ02] Ivan Damgård and Mads Jurik. "Client/Server Tradeoffs for Online Elections". In: *Public Key Cryptography*. Ed. by David Naccache and Pascal Paillier. Springer Berlin Heidelberg, 2002.

[Esc+20] Daniel Escudero et al. "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits". In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Springer, 2020, pp. 823–852.

[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: *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*. 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: *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'10. French Riviera, France: Springer-Verlag, 2010, pp. 506–522. ISBN: 3642131891. DOI: 10.1007/978-3-642-13190-5_26. URL: https://doi.org/10.1007/978-3-642-13190-5_26.

[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* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10.1145/116825.116852. URL: https://doi.org/10.1145/116825.116852.

[Her+95] Amir Herzberg et al. "Proactive Secret Sharing Or: How to Cope With Perpetual Leakage". In: *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 339–352. ISBN: 3540602216.

[HNP05a] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. "Cryptographic Asynchronous Multi-party Computation with Optimal Resilience". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 322–340.

[HNP05b] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. "Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract)". In: *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*. 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: (2014).

[Mar+19] Sai Krishna Deepak Maram et al. *CHURP: Dynamic-Committee Proactive Secret Sharing*. Cryptology ePrint Archive, Report 2019/017. https://eprint.iacr.org/2019/017. 2019.

[OY91] Rafail Ostrovsky and Moti Yung. "How to Withstand Mobile Virus Attacks (Extended Abstract)". In: *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '91. Montreal, Quebec, Canada: Association for Computing Machinery, 1991, pp. 51–59. ISBN: 0897914392. DOI: 10.1145/112600.112605. URL: https://doi.org/10.1145/112600.112605.

[Reg09] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *J. ACM* 56.6 (Sept. 2009). ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. URL: https://doi.org/10.1145/1568318.1568324.

[RSY18] Leonid Reyzin, Adam Smith, and Sophia Yakoubov. *Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC*. Cryptology ePrint Archive, Report 2018/997. https://eprint.iacr.org/2018/997. 2018.

[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.* 13.4 (Dec. 2010). ISSN: 1094-9224. DOI: 10.1145/1880022.1880028. URL: https://doi.org/10.1145/1880022.1880028.

[Sta96] Markus Stadler. "Publicly Verifiable Secret Sharing". In: *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*. 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 et al. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019.

[YY01] Adam L. Young and Moti Yung. "A PVSS as Hard as Discrete Log and Shareholder Separability". In: *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001,*

Cheju Island, Korea, February 13-15, 2001, Proceedings. Ed. by Kwangjo Kim. Lecture Notes in Computer Science. 2001.

[ZSR] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. "APSS: Proactive Secret Sharing in Asynchronous Systems". In: *In preparation* 8 (), p. 2005.

A REMINDER OF VERIFIABLE THRESHOLD ADDITIVE HOMOMORPHIC ENCRYPTION

We first recall the notion of threshold additive homomorphic encryption (AHE), as implemented in [CDN01; BHN10], at the cost of a *trusted setup*.

Definition 13. (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 negligible advantage, in λ, in distinguishing between the two distributions

$$\left\{c_m, m, \{\mu_i\}_{i\in\mathcal{I}}, Sim\big(c_m, m, \{\mu_i\}_{i\in\mathcal{I}}\big)\right\} \text{ and } \left\{c_m, m, \{\mu_i\}_{i\in\mathcal{I}}, \{\mu_i\}_{i\in[n]\setminus\mathcal{I}}\right\} \tag{8}$$

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 PSEUDOCODE OF THE COMPUTATION METHOD

B.0.1 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 2. Each message m is automatically stamped with $kingNb$, the king number that lead 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 aggregated contributions and a ***CP*** from $combine_{sid}$
- $VERIFCONTRIB$: message send by a slave that contains a partial signature of aggregated contributions from $sign$.

Combine Proof. A Combine Proof for a stage SID is a data type that contains the aggregation 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 aggregated result of this computation, and finally to the aggregated proof of correct computation. We note $sid.aggregate$ the aggregation 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 aggregates 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***.

B.1 Pseudocode of the structure of computation

The protocols are given in Algorithms 4 and 5. 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 2 gives utilities functions used by all parties to execute the protocol and algorithm 3 describes specific functions used by the king.

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) $m \leftarrow MSG(CONTRIBMSG, sid, \perp, \{m_i.sig\}_{m_i \in V}, \perp)$
(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) $m \leftarrow MSG(VERIFCONTRIB, m.sid.number, m.value, \perp, \perp)$
(2) $m.sig \leftarrow sign_j(m.kingNb, m.type, m.sid.number, , m.value, , m.proof)$
(3) **return** m

Fig. 2. Utilities

Lemma 14. *(Signature Phase) For Every possible $\mathcal{A}$ and for every possible scheduler, the Signature Phase achieves:* ***(1)*** *TERMINATION: All honest party will eventually terminate.* ***(2)*** *CORRECTNESS: For an honest king, the phase outputs a Quorum Verification Certificate.*

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

Fig. 3. King Utilities

Signature Phase

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

Fig. 4. Signature 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 $MSG(COMBMSG, sid, cp.value, m.qvc, cp.proof)$
(9) **as** a slave:
(10) Send to king $contrib_{sid}(\{m\}, s_j)$

Fig. 5. Contribution Phase

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 15. *(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 *Signature Phase* □

C RELATIONS TO BE PROVEN IN ZK FOR IMPLEMENTION OF TAE

The last relation is for the contribution to multiplication by a scalar λ, which is a special case of linear combination.

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

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

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

D PROOFS AND COMPLEMENTS OF TRIPLE GENERATION METHOD

D.1 TripVerif

Protocol $TripVerif(\{X^{(i)}, Y^{(i)}, Z^{(i)}\}_{i\in[n]}, \Pi)$

(1) The parties create an accumulative set $\mathcal{U}$.
(2) For all $i \in [n]$, the parties execute $MultVerif(X^{(i)}, Y^{(i)}, Z^{(i)}, \pi^{(i)})$. If all instances terminate for a given i, include P_i in $\mathcal{U}$.

Fig. 6. Triple verification

D.2 Proof of $TripTrans$

Transformation of independent multiplication triples to co-related triples:

Lemma 16. *Let $\{(A^{(j)}, B^{(j)}, C^{(j)})\}_{j\in[t+1+t']}$ be a set of $t+1+t'$ shared 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 $t+1+t'$ shared 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**) $\mathbb{Z}(.) = \mathbb{X}(.) \boxtimes \mathbb{Y}(.)$ [5] 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 17).

[5] For the rest of the proofs, $\boxtimes$ denotes the homomorphic multiplication

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 $\{\alpha_j, x^{(j)}\}_{j\in[\frac{t+t'}{2}+1]}$ and the points $\{\alpha_j, y^{(j)}\}_{j\in[\frac{t+t'}{2}+1]}$ define the polynomials $\mathbb{x}(.)$ and $\mathbb{y}(.)$ respectively of degree at most $(\frac{t+t'}{2})$.
(3) The 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 $\{\alpha_j, z^{(j)}\}_{j\in[t+1+t']}$ 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.

Fig. 7. Triple transforamtion

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

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 $z^{(i)}$[6].

□

D.2.1 *Proof of EncBeaver.*

Lemma 17. *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.*

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

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

- 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.

Fig. 8. EncBeaver

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

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

(1) The parties execute the protocol $TripTrans(\{(X^{(j)}, Y^{(lj)}, Z^{(j)})\}_{j\in[t+1+t']}$ 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 $\mathbf{A} = \mathbb{X}(\beta)$,$\mathbf{B} = \mathbb{Y}(\beta)$ and $\mathbf{C}= \mathbb{Z}(\beta)$ and terminate.

Fig. 9. Randomness extraction

Lemma 18. *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 output triple ($\mathbf{A} = \mathbb{X}(\beta)$,$\mathbf{B} = \mathbb{Y}(\beta)$ and $\mathbf{C} = \mathbb{Z}(\beta)$) is a multiplication triple **(3)** PRIVACY: The view of $\mathcal{A}$ in the protocol is distributed independently of the output multiplication triple $\{(\mathbf{A} = \mathbb{X}(\beta), \mathbf{B}= \mathbb{Y}(\beta), \mathbf{C} = \mathbb{Z}(\beta))\}$*

Proof. TERMINATION: This property directly follows from the termination property of the protocol $TripTrans$.
CORRECTNESS: To argue correctness, we have to show that the triple (**A**, **B**, **C**) is a valid multiplication triple. We recall that ($\mathbf{A} = \mathbb{X}(\beta)$,$\mathbf{B} = \mathbb{Y}(\beta)$ and $\mathbf{C} = \mathbb{Z}(\beta)$). To complete the proof, it is enough to show that the protocol ensures the multiplicative relation $\mathbb{Z}(.) = \mathbb{X}(.) \boxtimes \mathbb{Y}(.)$ 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 triple (**A**, **B**, **C**). 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 ($\mathbb{X}(\gamma)$ =**A**) and with the prior knowledge of $\mathcal{A}$. Thus, $\mathbb{X}(\beta) = \mathbf{A}$ will be random to $\mathcal{A}$. The same argument

allows us to claim that **B** and **C** will be random to $\mathcal{A}$ subject to $\mathbb{Z}(\beta) = \mathbb{X}(\beta) \boxtimes \mathbb{Y}(\beta)$. The security property of the encryption scheme allows us to claim that (**a**, **b**, **c**) are unknown to $\mathcal{A}$. □

D.3 The Preprocessing phase protocol

Protocol *PreProc*

First synchronous broadcast round

(1) **Triple sharing** - For $i \in [n]$, evey party P_i executes the following code:
- Act as a dealer D and execute and instance of *Share* to share 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 execute $TripVerif(\{(X^{(i)}, Y^{(i)}, Z^{(i)}\}_{i\in[n]})$ and output a set $\mathcal{U}$ consisting of $t + 1 + t'$ parties

(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 a triple $\{(\mathbf{A}, \mathbf{B}, \mathbf{C})\}$ and terminate.

Fig. 10. Preprocessing overview

Lemma 19. *For every possible $\mathcal{A}$ and every possible scheduler, protocol PreProc achieves:* ***(1)*** *TERMINATION: All honest parties terminate the protocol.* ***(2)*** *CORRECTNESS: The output triple will be multiplication triple.* ***(3)*** *PRIVACY: The output triple is 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 $TripVerif$ and $TripExt$.

PRIVACY: Given that there will be at least $t + 1$ honest parties in set $\mathcal{U}$ and that the multiplication triples shared 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 $\mathcal{A}$.

□