Applications of Secure Multiparty Computation
P. Laud and L. Kamm (Eds.)

doi:10.3233/978-1-61499-532-6-165

Chapter 9
Verifiable Computation in Multiparty Protocols with Honest Majority

Alisa PANKOVA [a] and Peeter LAUD [a]
[a] *Cybernetica AS, Estonia*

Abstract. We present a generic method for turning passively secure protocols into protocols secure against covert attacks. This method adds to the protocol a post-execution verification phase that allows a misbehaving party to escape detection only with negligible probability. The execution phase, after which the computed protocol result is already available to the parties, has only negligible overhead added by our method.

The method uses shared verification based on linear probabilistically checkable proofs. The checks are done in zero-knowledge, thereby preserving the privacy guarantees of the original protocol. This method is inspired by recent results in verifiable computation, adapting them to the multiparty setting and significantly lowering their computational costs for the provers. The verification is straightforward to apply to protocols over finite fields.

A longer preprocessing phase can be introduced to shorten the verification phase even more. Beaver triples can be used to make it possible to verify the entire protocol execution locally on shares, leaving for verification just some linear combinations that do not need complex zero-knowledge proofs. Using preprocessing provides a natural way of verifying computation over rings of the size of 2^n.

Introduction

Any multiparty computation can be performed so that the participants only learn their own outputs and nothing else [1]. While the generic construction is expensive in computation and communication, the result has sparked research activities in secure multiparty computation (SMC), with results that are impressive both performance-wise [2,3,4,5] as well as in the variety of concrete problems that have been tackled [6,7,8,9]. From the start, two kinds of adversaries — passive and active — have been considered in the construction of SMC protocols, with the highest performance and the greatest variety achieved for protocol sets secure against passive adversaries.

Verifiable computation (VC) [10] allows a weak client to outsource a computation to a more powerful server that accompanies the computed result with proof of correct computation, the verification of which by the client is cheaper than performing the computation itself. VC can be used to strengthen protocols secure against passive adversaries — after executing the protocol, the parties can prove to each other that they have correctly followed the protocol. If the majority of the parties are honest (an assumption which is made also by the most efficient SMC protocol sets secure against passive adversaries),

then the resulting protocol will satisfy a strong version of *covert security* [11], where any deviations from the protocol are guaranteed to be discovered and reported. Unfortunately, existing approaches to VC have a large computational overhead for the server/prover. Typically, if the computation is represented as an arithmetic circuit C, the prover has to perform $\Omega(|C|)$ public-key operations in order to ensure its good behavior, as well as to protect its privacy.

In this work we show that in the multiparty context with an honest majority, these public-key operations are not necessary. Instead, verifications can be done in a distributed manner, in a way that provides the same security properties. For this, we apply the ideas of existing VC approaches based on linear probabilistically checkable proofs (PCPs) [12], and combine them with linear secret sharing, which we use also for commitments. We end up with a protocol transformation that makes the executions of any protocol (and not just SMC protocols) verifiable afterwards. Our transformation commits the randomness (this takes place offline), inputs, and the communication of the participants. The commitments are cheap, as they are based on digital signatures, and do not add a significant overhead to the execution phase. The results of the protocol are available after the execution. The verification can take place at any time after the execution. Dedicated high-bandwidth high-latency communication channels can be potentially used for it. The verification itself is succinct. The proof is generated in $O(|C| \log |C|)$ field operations, but the computation is local. The generation of challenges costs $O(1)$ in communication and $O(|C|)$ in local computation.

We present our protocol transformation as a functionality in the universal composability (UC) framework, which is described more precisely in Chapter 1. After reviewing related work in Sec. 1, we describe the ideal functionality in Sec. 2 and its implementation in Sec. 4. Before the latter, we give an overview of the existing building blocks we use in Sec. 3. We estimate the computational overhead of our transformation in Sec. 5.

Apart from increasing the security of SMC protocols, our transformation can be used to add verifiability to other protocols. In Sec. 6 we demonstrate how a verifiable secret sharing (VSS) scheme can be constructed. We compare it with state-of-the-art VSS schemes and find that despite much higher generality, our construction enjoys similar complexity.

In order to make the verification phase even more efficient, in Sec. 7 we propose to push more computation into the preprocessing phase. This allows to simplify the final zero-knowledge proofs significantly.

1. Related Work

The property brought by our protocol transformation is similar to security against *covert* adversaries [11] that are prevented from deviating from the prescribed protocol by a non-negligible chance of getting caught.

A similar transformation, applicable to protocols of a certain structure, was introduced by Damgård et al. [13]. They run several instances of the initial protocol, where only one instance is run on real inputs, and the other on randomly generated shares. No party should be able to distinguish the protocol executed on real inputs from the protocol executed on random inputs. In the end, the committed traces of the random executions are revealed by each party, and everyone may check if a party acted honestly in the

random executions. This way, in the beginning all the inputs must be reshared, and the computation must leak no information about the inputs, so that no party can guess which inputs are real and which are random. Hence, the transformation does not allow using the advantage gained from specific sharings of inputs between the parties (where a party can recognize its input), or deliberated leakage of the information that will be published in the end anyway. The probability of cheating decreases linearly with the number of dummy executions.

Compared to the transformation of [13], ours is more general, has lower overhead in the execution phase, and is guaranteed to catch the deviating parties. Our transformation can handle protocols where some of the results are made available to the computing parties already before the end of the protocol. This may significantly decrease the complexity of the protocol [8]. A good property of their construction is its black box nature, which our transformation does not have. Hence, different transformations may be preferable in different situations.

Many works have been dedicated to short verifications of solutions to NP-complete problems. Probabilistically checkable proofs [14] allow verifying a possibly long proof by querying a small number of its bits. Micali [15] has presented *computationally sound proofs* where the verification is not perfect, and the proof can be forged, but it is computationally hard to do. Kilian [16] proposed interactive probabilistically checkable proofs using bit commitments. A certain class of *linear* probabilistically checkable proofs [12] allows making argument systems much simpler and more general.

In computation verification, the prover has to prove that given the valuations of certain wires of a circuit, there is a correct valuation of all the other wires so that the computation is correct with respect to the given circuit. Verifiable computation can in general be based not only on the PCP theorem. In [10], Yao's garbled circuits [17] are executed using fully homomorphic encryption (see Chapter 1 for details). Quadratic span programs for Boolean circuits and quadratic arithmetic programs for arithmetic circuits without PCP have first been proposed in [18], later extended to PCP by [19], and further optimized and improved in [20,21,22]. Particular implementations of verifiable computations have been done, for example, in [21,22,23].

The goal of our transformation is to provide security against a certain form of active attackers. SMC protocols secure against active attackers have been known for a long time [1,24]. SPDZ [4,25] is currently the SMC protocol set secure against active adversaries with the best online performance achieved through extensive offline precomputations (see Chapter 1 for details of SPDZ protocols). Similarly to several other protocol sets, SPDZ provides only a minimum amount of protocols to cooperatively evaluate an arithmetic circuit. We note that very recently, a form of post-execution verifiability has been proposed for SPDZ [26].

2. Ideal Functionality

We use the universal composability (UC) framework [27] to specify our verifiable execution functionality. We have n parties (indexed by $[n] = \{1, \ldots, n\}$), where $\mathcal{C} \subseteq [n]$ are corrupted for $|\mathcal{C}| = t < n/2$ (we denote $\mathcal{H} = [n] \backslash \mathcal{C}$). The protocol has r rounds, where the ℓ-th round computations of the party P_i, the results of which are sent to the party P_j, are given by an arithmetic circuit C_{ij}^{ℓ}, either over a finite field $\mathbb{F}$ or over rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$. We define the following gate operations for such a circuit:

In the beginning, $\mathcal{F}_{vmpc}$ gets from $\mathcal{Z}$ for each party P_i the message $(\mathsf{circuits}, i, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ and forwards them all to $\mathcal{A}_S$. For each $i \in \mathcal{H}$ [resp $i \in \mathcal{C}$], $\mathcal{F}_{vmpc}$ gets $(\mathsf{input}, \mathbf{x}_i)$ from $\mathcal{Z}$ [resp. $\mathcal{A}_S$]. For each $i \in [n]$, $\mathcal{F}_{vmpc}$ randomly generates $\mathbf{r}_i$. For each $i \in \mathcal{C}$, it sends $(\mathsf{randomness}, i, \mathbf{r}_i)$ to $\mathcal{A}_S$.
For each round $\ell \in [r]$, $i \in \mathcal{H}$ and $j \in [n]$, $\mathcal{F}_{vmpc}$ uses C_{ij}^{ℓ} to compute the message $\mathbf{m}_{ij}^{\ell}$. For all $j \in \mathcal{C}$, it sends $\mathbf{m}_{ij}^{\ell}$ to $\mathcal{A}_S$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\mathbf{m}_{ji}^{\ell}$ from $\mathcal{A}_S$.
After r **rounds**, $\mathcal{F}_{vmpc}$ sends $(\mathsf{output}, \mathbf{m}_{1i}^{r}, \ldots, \mathbf{m}_{ni}^{r})$ to each party P_i with $i \in \mathcal{H}$. Let $r' = r$ and $\mathcal{B}_0 = \emptyset$.
Alternatively, **at any time** before outputs are delivered to parties, $\mathcal{A}_S$ may send $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$, with $\mathcal{B}_0 \subseteq \mathcal{C}$. In this case the outputs are not sent. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.
After r' **rounds**, $\mathcal{A}_S$ sends to $\mathcal{F}_{vmpc}$ the messages $\mathbf{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$.
$\mathcal{F}_{vmpc}$ defines $\mathcal{M} = \mathcal{B}_0 \cup \{i \in \mathcal{C} \mid \exists j \in [n], \ell \in [r'] : \mathbf{m}_{ij}^{\ell} \neq C_{ij}^{\ell}(\mathbf{x}_i, \mathbf{r}_i, \mathbf{m}_{1i}^{1}, \ldots, \mathbf{m}_{ni}^{\ell-1})\}$.
Finally, for each $i \in \mathcal{H}$, $\mathcal{A}_S$ sends $(\mathsf{blame}, i, \mathcal{B}_i)$ to $\mathcal{F}_{vmpc}$, with $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. $\mathcal{F}_{vmpc}$ forwards this message to P_i.

Figure 1. The ideal functionality for verifiable computations

- The operations $+$ (addition) and $*$ (multiplication) are in $\mathbb{F}$ or in rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$ ($n_i < n_j$ for $i < j$).
- The operations trunc and zext are between rings. Let $x \in \mathbb{Z}_{n_x}$, $y \in \mathbb{Z}_{n_y}$, $n_x < n_y$.
  - $x = \mathsf{trunc}(y)$ computes $x = y \mod n_x$, going from a larger ring to a smaller ring.
  - $y = \mathsf{zext}(x)$ takes x and uses the same value in n_y. It can be treated as taking the bits of x and extending them with zero bits.
- The operation bits are from an arbitrary ring $\mathbb{Z}_n$ to $(\mathbb{Z}_2)^{\log n}$. This operation performs a bit decomposition. Although bit decomposition can be performed by other means, we introduce a separate operation as it is reasonable to implement a faster verification for it.

More explicit gate types can be added to the circuit. Although the current set of gates is sufficient to represent any other operation, the verifications designed for special gates may be more efficient. For example, introducing the division gate $c = a/b$ explicitly would allow to verify it as $a = b * c$ instead of expressing the division through addition and multiplication. In this work, we do not define any other gates, as the verification of most standard operations is fairly straightforward, assuming that bit decomposition is available.

The circuit C_{ij}^{ℓ} computes the ℓ-th round messages $\mathbf{m}_{ij}^{\ell}$ to all parties $j \in [n]$ from the input $\mathbf{x}_i$, randomness $\mathbf{r}_i$ and the messages P_i has received before (all values $\mathbf{x}_i, \mathbf{r}_i, \mathbf{m}_{ij}^{\ell}$ are vectors over rings $\mathbb{Z}_n$ or a finite field $\mathbb{F}$). We define that the messages received during the r-th round comprise the *output of the protocol*. The ideal functionality $\mathcal{F}_{vmpc}$, running in parallel with the environment $\mathcal{Z}$ and the adversary $\mathcal{A}_S$, is given in Fig. 1.

We see that $\mathcal{M}$ is the set of all parties that deviate from the protocol. Our verifiability property is very strong as *all* of them will be reported to *all* honest parties. Even if only *some* rounds of the protocol are computed, all the parties that deviated from the protocol in completed rounds will be detected. Also, no honest parties (in $\mathcal{H}$) can be falsely blamed. We also note that if $\mathcal{M} = \emptyset$, then $\mathcal{A}_S$ does not learn anything that a semi-honest adversary could not learn.

$\mathcal{F}_{transmit}$ works with unique message identifiers mid, encoding a sender $s(mid) \in [n]$, a receiver $r(mid) \in [n]$, and a party $f(mid) \in [n]$ to whom the message should be forwarded by the receiver (if no forwarding is foreseen then $f(mid) = r(mid)$).
Secure transmit: Receiving $(\mathsf{transmit}, mid, m)$ from $P_{s(mid)}$ and $(\mathsf{transmit}, mid)$ from all (other) honest parties, store $(mid, m, r(mid))$, mark it as undelivered, and output $(mid, |m|)$ to the adversary. If the input of $P_{s(mid)}$ is invalid (or there is no input), and $P_{r(mid)}$ is honest, then output $(\mathsf{corrupt}, s(mid))$ to all parties.
Secure broadcast: Receiving $(\mathsf{broadcast}, mid, m)$ from $P_{s(mid)}$ and $(\mathsf{broadcast}, mid)$ from all honest parties, store (mid, m, bc), mark it as undelivered, output $(mid, |m|)$ to the adversary. If the input of $P_{s(mid)}$ is invalid, output $(\mathsf{corrupt}, s(mid))$ to all parties.
Synchronous delivery: At the end of each round, for each undelivered (mid, m, r) send (mid, m) to P_r; mark (mid, m, r) as delivered. For each undelivered (mid, m, bc), send (mid, m) to each party and the adversary; mark (mid, m, bc) as delivered.
Forward received message: On input $(\mathsf{forward}, mid)$ from $P_{r(mid)}$ after (mid, m) has been delivered to $P_{r(mid)}$, and receiving $(\mathsf{forward}, mid)$ from all honest parties, store $(mid, m, f(mid))$, mark as undelivered, output $(mid, |m|)$ to the adversary. If the input of $P_{r(mid)}$ is invalid, and $P_{f(mid)}$ is honest, output $(\mathsf{corrupt}, r(mid))$ to all parties.
Publish received message: On input $(\mathsf{publish}, mid)$ from the party $P_{f(mid)}$, which at any point received (mid, m), output (mid, m) to each party, and also to the adversary.
Do not commit corrupt to corrupt: If for some mid both $P_{s(mid)}, P_{r(mid)}$ are corrupt, then on input $(\mathsf{forward}, mid)$ the adversary can ask $\mathcal{F}_{transmit}$ to output (mid, m') to $P_{f(mid)}$ for any m'. If, additionally, $P_{f(mid)}$ is corrupt, then the adversary can ask $\mathcal{F}_{transmit}$ to output (mid, m') to all honest parties.

Figure 2. Ideal functionality $\mathcal{F}_{transmit}$

3. Building Blocks

Throughout this work, bold letters $\mathbf{x}$ denote vectors, where x_i denotes the i-th coordinate of $\mathbf{x}$. Concatenation of $\mathbf{x}$ and $\mathbf{y}$ is denoted by $(\mathbf{x}\|\mathbf{y})$, and their scalar product by $\langle \mathbf{x}, \mathbf{y} \rangle$, which is defined (only if $|\mathbf{x}| = |\mathbf{y}|$) as $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{|\mathbf{x}|} x_i y_i$. Our implementation uses a number of previously defined subprotocols and algorithm sets.

Message transmission. For message transmission between parties, we use functionality $\mathcal{F}_{transmit}$ [13], which allows one to provably reveal to third parties the messages that one received during the protocol, and to further transfer such revealed messages. Our definition of $\mathcal{F}_{transmit}$ differs from that of Damgård et al. [13] $\mathcal{F}_{transmit}$ by supporting the forwarding of received messages as well as broadcasting as a part of the outer protocol. The definition of the ideal functionality of $\mathcal{F}_{transmit}$ is shown in Fig. 2. The real implementation of the transmission functionality is built on top of signatures. This makes the implementation very efficient, as hash trees allow several messages (sent in the same round) to be signed with almost the same computation effort as a single one [28], and signatures can be verified in batches [29]. An implementation of $\mathcal{F}_{transmit}$ is given in [30].

Shamir's secret sharing. For commitments, we use (n,t) Shamir's secret sharing [31], where any t parties are able to recover the secret, but fewer than t are not (see Chapter 1 for details). By sharing a *vector* $\mathbf{x}$ over $\mathbb{F}$ into vectors $\mathbf{x}^1, \ldots, \mathbf{x}^n$ we mean that each i-th entry $x_i \in \mathbb{F}$ of $\mathbf{x}$ is shared into the i-th entries $x_i^1 \in \mathbb{F}, \ldots, x_i^n \in \mathbb{F}$ of $\mathbf{x}^1, \ldots, \mathbf{x}^n$. In this way, for each $T = \{i_1, \ldots, i_t\} \subseteq [n]$, the entries can be restored as $x_i = \sum_{j=1}^{t} b_{Tj} x_i^{i_j}$ for certain

constants b_{Tj}, and hence, $\mathbf{x} = \sum_{j=1}^{t} b_{Tj}\mathbf{x}^{i_j}$. The linearity extends to scalar products: if a vector π is shared to $\pi^1,\ldots,\pi^n$, then for any vector $\mathbf{q}$ and $T = \{i_1,\ldots,i_t\}$, we have $\sum_{j=1}^{t} b_{Tj}\langle\pi^{i_j},\mathbf{q}\rangle = \langle\pi,\mathbf{q}\rangle$.

We note that sharing a value x as $x^1 = \cdots = x^k = x$ is valid, i.e. x can be restored from $x^{i_1},\ldots,x^{i_t}$ by forming the same linear combination. In our implementation of the verifiable computation functionality, we use such sharing for values that become public due to the adversary's actions.

Beaver triples. This primitive will be used in cases where we want to make the verification phase more efficient by pushing some computation into the preprocessing phase. Beaver triples [32] are triples of values (a,b,c) in a ring $\mathbb{Z}_n$, so that $a,b \xleftarrow{\$} R$, and $c = a \cdot b$. Precomputing such triples can be used to linearize multiplications. For example, if we want to multiply $x \cdot y$, and a triple (r_x, r_y, r_{xy}) is already precomputed and preshared, we may first compute and publish $x' := x - r_x$ and $y' := y - r_y$ (x' and y' leak no information about x and y), and then compute the linear combination $x \cdot y = (x' + r_x)(y' + r_y) = x'y' + r_x y' + x' r_y + r_x r_y = x'y' + r_x y' + x' r_y + r_{xy}$. Differently from standard usage (like in SPDZ), we do not use these triples in the original protocol, but instead use them to simplify the verification phase. See Chapter 1 for details on the generation and usage of such triples.

Linear PCP. This primitive forms the basis of our verification. Before giving its definition, let us formally state when a protocol is statistically privacy-preserving.

Definition 1 (δ-private protocol [33]) *Let Π be a multiparty protocol that takes input $\mathbf{x}$ from honest parties and $\mathbf{y}$ from adversarially controlled parties. The protocol Π is* δ-private against a class of adversaries $\mathcal{A}$ *if there exists a simulator* Sim*, so that for all adversaries $A \in \mathcal{A}$ and inputs $\mathbf{x},\mathbf{y}$, $\left|\mathbf{Pr}\left[A^{\Pi(\mathbf{x},\mathbf{y})}(\mathbf{y}) = 1\right] - \mathbf{Pr}\left[A^{\mathsf{Sim}(\mathbf{y})}(\mathbf{y}) = 1\right]\right| \leq \delta$.*

Definition 2 (Linear probabilistically checkable proof (LPCP) [19]) *Let $\mathbb{F}$ be a finite field, $\upsilon,\omega \in \mathbb{N}$, $R \subseteq \mathbb{F}^{\upsilon} \times \mathbb{F}^{\omega}$. Let P and Q be probabilistic algorithms, and D a deterministic algorithm. The pair (P,V), where $V = (Q,D)$ is a d-query* δ-statistical honest verifier zero-knowledge (HVZK) linear PCP for R with the knowledge error ε and the query length m*, if the following holds.*

Syntax *On input $\mathbf{v} \in \mathbb{F}^{\upsilon}$ and $\mathbf{w} \in \mathbb{F}^{\omega}$, algorithm P computes $\pi \in \mathbb{F}^m$. The algorithm Q randomly generates d vectors $\mathbf{q}_1,\ldots\mathbf{q}_d \in \mathbb{F}^m$ and some state information $\mathbf{u}$. On input $\mathbf{v}$, $\mathbf{u}$, as well as $a_1,\ldots,a_d \in \mathbb{F}$, the algorithm D accepts or rejects. Let $V^{\pi}(\mathbf{v})$ denote the execution of Q followed by the execution of V on $\mathbf{v}$, the output $\mathbf{u}$ of Q, and $a_1,\ldots,a_d$, where $a_i = \langle\pi,\mathbf{q}_i\rangle$.*

Completeness *For every $(\mathbf{v},\mathbf{w}) \in R$, the output of $P(\mathbf{v},\mathbf{w})$ is a vector $\pi \in \mathbb{F}^m$ so that $V^{\pi}(\mathbf{v})$ accepts with probability 1.*

Knowledge *There is a knowledge extractor E so that for every vector $\pi^* \in \mathbb{F}^m$, if $\Pr\left[V^{\pi^*}(\mathbf{v})\ accepts\right] \geq \varepsilon$ then $E(\pi^*,\mathbf{v})$ outputs $\mathbf{w}$ so that $(\mathbf{v},\mathbf{w}) \in R$.*

Honest Verifier Zero-Knowledge *The protocol between an honest prover executing $\pi \leftarrow P(\mathbf{v},\mathbf{w})$ and an adversarial verifier executing $V^{\pi}(\mathbf{v})$ with common input $\mathbf{v}$ and the prover's input $\mathbf{w}$ is δ-private for the class of passive adversaries.*

Similarly to different approaches to verifiable computation [18,19,20,21,22], in our work we let the relation R correspond to the circuit C executed by the party whose observance of the protocol is being verified. In this correspondence, $\mathbf{v}$ is the tuple of all inputs, outputs, and the used random values of that party. The vector $\mathbf{w}$ extends $\mathbf{v}$ with the results of all intermediate computations by that party. Differently from existing approaches, $\mathbf{v}$ itself is private. Hence, it is unclear how the decision algorithm D can be executed on it. Therefore, we do not use D as a black box, but build our solution on top of a particular LPCP [21].

The LPCP algorithms used by Ben-Sasson et al. [21] are statistical HVZK. Namely, the values $\langle \pi, \mathbf{q}_i \rangle$ do not reveal any private information about π, unless the random seed $\tau \in \mathbb{F}$ for Q is chosen in badly, which happens with negligible probability for a sufficiently large field. In [21], Q generates 5 challenges $\mathbf{q}_1, \ldots, \mathbf{q}_5$ and the state information $\mathbf{u}$ with the length $|\mathbf{v}| + 2$. Given the query results $a_i = \langle \pi, \mathbf{q}_i \rangle$ for $i \in \{1, \ldots, 5\}$ and the state information $\mathbf{u} = (u_0, u_1, \ldots, u_{|\mathbf{v}|+1})$, the following two checks have to pass:

$$a_1 a_2 - a_3 - a_4 u_{|\mathbf{v}|+1} = 0, \qquad (*)$$
$$a_5 - \langle (1 \| \mathbf{v}), (u_0, u_1, \ldots, u_{|\mathbf{v}|}) \rangle = 0. \qquad (**)$$

Here $(*)$ is used to show the existence of $\mathbf{w}$, and $(**)$ shows that a certain segment of π equals $(1\|\mathbf{v})$ [21]. Throughout this work, we reorder the entries of π compared to [21] and write $\pi = (\mathbf{p}\|1\|\mathbf{v})$, where $\mathbf{p}$ represents all the other entries of π, as defined in [21]. The challenges $\mathbf{q}_1, \ldots, \mathbf{q}_5$ are reordered in the same way.

This linear interactive proof can be converted into a zero-knowledge succinct non-interactive argument of knowledge [19]. Unfortunately, it requires homomorphic encryption, and the number of encryptions is linear in the size of the circuit. We show that the availability of honest majority allows the proof to be completed without public-key encryptions.

The multiparty setting introduces a further difference from [21]: the vector $\mathbf{v}$ can no longer be considered public, as it contains a party's private values. Thus, we have to strengthen the HVZK requirement in Def. 2, making $\mathbf{v}$ private to the prover. The LPCP constructions of [21] do not satisfy this strengthened HVZK requirement, but their authors show that this requirement would be satisfied if a_5 were not present. In the following, we propose a construction where just the first check $(*)$ is sufficient, so only $a_1, \ldots, a_4$ have to be published. We prove that the second check $(**)$ will be passed implicitly. We show the following.

Theorem 1 *Given a δ-statistical HVZK instance of the LPCP of Ben-Sasson et al. [21] with the knowledge error ε, any n-party r-round protocol Π can be transformed into an n-party $(r+8)$-round protocol Ξ in the $\mathcal{F}_{transmit}$-hybrid model, which computes the same functionality as Π and achieves covert security against adversaries statically corrupting at most $t < n/2$ parties, where the cheating of any party is detected with probability of at least $(1-\varepsilon)$. If Π is δ'-private against passive adversaries statically corrupting at most t parties, then Ξ is $(\delta' + \delta)$-private against cover adversaries. Under active attacks by at most t parties, the number of rounds of the protocol may at most double.*

Thm. 1 is proved by the construction of the real functionality Sec. 4, as well as the simulator presented in [30]. In the construction, we use the following algorithms implicitly defined by Ben-Sasson et al. [21]:

Circuits: M_i gets from $\mathcal{Z}$ the message $(\mathsf{circuits}, i, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ and sends it to $\mathcal{A}$.
Randomness generation and commitment: Let $\mathcal{R} = [t+1]$. For all $i \in \mathcal{R}$, $j \in [n]$, M_i generates $\mathbf{r}_{ij}$ for M_j. M_i shares $\mathbf{r}_{ij}$ to n vectors $\mathbf{r}_{ij}^1, \ldots, \mathbf{r}_{ij}^n$ according to $(n, t+1)$ Shamir's scheme. For $j \in [n]$, M_i sends $(\mathsf{transmit}, (\mathsf{r\_share}, i, j, k), \mathbf{r}_{ij}^k)$ to $\mathcal{F}_{transmit}$ for M_k.
Randomness approval: For each $j \in [n] \setminus \{k\}$, $i \in \mathcal{R}$, M_k sends $(\mathsf{forward}, (\mathsf{r\_share}, i, j, k))$ to $\mathcal{F}_{transmit}$ for M_j. Upon receiving $((\mathsf{r\_share}, i, j, k), \mathbf{r}_{ij}^k)$ for all $k \in [n]$, $i \in \mathcal{R}$, M_j checks if the shares comprise a valid $(n, t+1)$ Shamir's sharing. M_j sets $\mathbf{r}_i = \sum_{i \in \mathcal{R}} \mathbf{r}_{ij}$.
Input commitments: M_i with $i \in \mathcal{H}$ [resp. $i \in \mathcal{C}$] gets from $\mathcal{Z}$ [resp. $\mathcal{A}$] the input $\mathbf{x}_i$ and shares it to n vectors $\mathbf{x}_i^1, \ldots, \mathbf{x}_i^n$ according to $(n, t+1)$ Shamir's scheme. For each $k \in [n] \setminus \{i\}$, M_i sends to $\mathcal{F}_{transmit}$ $(\mathsf{transmit}, (\mathsf{x\_share}, i, k), \mathbf{x}_i^k)$ for M_k.
At any time: If $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$, M_i writes $mlc_i[j] := 1$ and goes to the accusation phase.

Figure 3. Initialization phase of the real functionality

- *witness*$(C, \mathbf{v})$: if $\mathbf{v}$ corresponds to a valid computation of C, it returns a witness $\mathbf{w}$ so that $(\mathbf{v}, \mathbf{w}) \in \mathcal{R}_C$.
- *proof*$(C, \mathbf{v}, \mathbf{w})$ if $(\mathbf{v}, \mathbf{w}) \in \mathcal{R}_C$, it constructs a corresponding proof $\mathbf{p}$.
- *challenge*(C, τ): returns $\mathbf{q}_1, \ldots, \mathbf{q}_5, \mathbf{u}$ that correspond to τ, so that:
  - for any valid proof $\pi = (\mathbf{p} \| 1 \| \mathbf{v})$, where $\mathbf{p}$ is generated by *proof*$(C, \mathbf{v}, \mathbf{w})$ for $(\mathbf{v}, \mathbf{w}) \in \mathcal{R}_C$, the checks $(*)$ and $(**)$ succeed with probability 1;
  - for any proof π^* generated without knowing τ, or such $\mathbf{w}$ that $(\mathbf{v}, \mathbf{w}) \in \mathcal{R}_C$, either $(*)$ or $(**)$ fails, except with negligible probability ε.

4. Real Functionality

Our initial construction works for arithmetic circuits over a finite field $\mathbb{F}$, assuming that the only gate types are $+$ and $*$. In Sec. 7, we show how it can be extended to a circuit over multiple rings, so that the gates trunc, zext, and bits can be added.

The protocol Π_{vmpc} implementing $\mathcal{F}_{vmpc}$ consists of n machines $M_1, \ldots, M_n$ doing the work of parties $P_1, \ldots, P_n$, and the functionality $\mathcal{F}_{transmit}$. The internal state of each M_i contains a bit-vector mlc_i of the length n, where M_i marks which other parties are acting maliciously. The protocol Π_{vmpc} runs in five phases: initialization, execution, message commitment, verification, and accusation.

In the initialization phase, the inputs $\mathbf{x}_i$ and the randomness $\mathbf{r}_i$ are committed. It is ensured that the randomness indeed comes from uniform distribution. This phase is given in Fig.3. If at any time $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$, each (uncorrupted) M_i writes $mlc_i[j] := 1$ (for each message $(\mathsf{corrupt}, j)$) and goes to the accusation phase.

In the execution phase, the parties run the original protocol as before, just using $\mathcal{F}_{transmit}$ to exchange the messages. This phase is given in Fig.4. If at any time in some round ℓ the message $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$ (all uncorrupted machines receive it at the same time), the execution is cut short, no outputs are produced and the protocol continues with the commitment phase.

In the message commitment phase, all the n parties finally commit their sent messages $\mathbf{c}_{ij}^{\ell}$ for each round $\ell \in [r']$ by sharing them to $\mathbf{c}_{ij}^{\ell 1}, \ldots, \mathbf{c}_{ij}^{\ell n}$ according to $(n, t+1)$ Shamir's scheme. This phase is given in Fig. 5. Let $\mathbf{v}_{ij}^{\ell} = (\mathbf{x}_i \| \mathbf{r}_i \| \mathbf{c}_{1i}^1 \| \cdots \| \mathbf{c}_{ni}^{\ell-1} \| \mathbf{c}_{ij}^{\ell})$ be the

For each round ℓ the machine M_i computes $\mathbf{c}_{ij}^{\ell} = C_{ij}^{\ell}(\mathbf{x}_i, \mathbf{r}_i, \mathbf{c}_{1i}^{1}, \ldots, \mathbf{c}_{ni}^{\ell-1})$ for each $j \in [n]$ and sends to $\mathcal{F}_{transmit}$ the message $(\mathsf{transmit}, (\mathsf{message}, \ell, i, j), \mathbf{c}_{ij}^{\ell})$ for M_j.
After r **rounds**, uncorrupted M_i sends $(\mathsf{output}, \mathbf{c}_{1i}^{r}, \ldots, \mathbf{c}_{ni}^{r})$ to $\mathcal{Z}$ and sets $r' := r$.
At any time: If $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$, each (uncorrupted) M_i writes $mlc_i[j] := 1$, sets $r' := \ell - 1$ and goes to the message commitment phase.

Figure 4. Execution phase of the real functionality

Message sharing: As a sender, M_i shares $\mathbf{c}_{ij}^{\ell}$ to $\mathbf{c}_{ij}^{\ell 1}, \ldots, \mathbf{c}_{ij}^{\ell n}$ according to $(n, t+1)$ Shamir's scheme. For each $k \in [n] \setminus \{i\}$, M_i sends to $\mathcal{F}_{transmit}$ the messages $(\mathsf{transmit}, (\mathsf{c\_share}, \ell, i, j, k), \mathbf{c}_{ij}^{\ell k})$ for M_j.
Message commitment: Upon receiving $((\mathsf{c\_share}, \ell, i, j, k), \mathbf{c}_{ij}^{\ell k})$ from $\mathcal{F}_{transmit}$ for all $k \in [n]$, the machine M_j checks if the shares correspond to the $\mathbf{c}_{ij}^{\ell}$ it has already received. If they do not, M_j sends $(\mathsf{publish}, (\mathsf{message}, \ell, i, j))$ to $\mathcal{F}_{transmit}$, so now everyone sees the values that it has actually received from M_i, and each (uncorrupted) M_k should now use $\mathbf{c}_{ij}^{\ell k} := \mathbf{c}_{ij}^{\ell}$. If the check succeeds, then M_i sends to $\mathcal{F}_{transmit}$ $(\mathsf{forward}, (\mathsf{c\_share}, \ell, i, j, k))$ for M_k for all $k \in [n] \setminus \{i\}$.

Figure 5. Message commitment phase of the real functionality

vector of inputs and outputs to the circuit C_{ij}^{ℓ} that M_i uses to compute the ℓ-th message to M_j. If the check performed by M_j fails, then M_j has received from M_i enough messages to prove its corruptness to others (but Fig. 5 presents an alternative, by publicly agreeing on $\mathbf{c}_{ij}^{\ell}$). After this phase, M_i has shared $\mathbf{v}_{ij}^{\ell}$ among all n parties. Let $\mathbf{v}_{ij}^{\ell k}$ be the share of $\mathbf{v}_{ij}^{\ell}$ given to machine M_k.

Each M_i generates a witness $\mathbf{w}_{ij}^{\ell} = witness(C_{ij}^{\ell}, \mathbf{v}_{ij}^{\ell})$, a proof $\mathbf{p}_{ij}^{\ell} = proof(C_{ij}^{\ell}, \mathbf{v}_{ij}^{\ell}, \mathbf{w}_{ij}^{\ell})$, and $\pi_{ij}^{\ell} = (\mathbf{p}_{ij}^{\ell} \| 1 \| \mathbf{v}_{ij}^{\ell})$ in the verification phase, as explained in Sec. 3. The vector $\mathbf{p}_{ij}^{\ell}$ is shared to $\mathbf{p}_{ij}^{\ell 1}, \ldots, \mathbf{p}_{ij}^{\ell n}$ according to $(n, t+1)$ Shamir's scheme.

All parties agree on a random τ, with M_i broadcasting τ_i and τ being their sum. A party refusing to participate is ignored. The communication must be synchronous, with no party P_i learning the values τ_j from others before he has sent his own τ_i. Note that $\mathcal{F}_{transmit}$ already provides this synchronicity. If it were not available, then standard tools (commitments) could be used to achieve fairness.

All (honest) parties generate $\mathbf{q}_{1ij}^{\ell}, \ldots, \mathbf{q}_{4ij}^{\ell}, \mathbf{q}_{5ij}^{\ell}, \mathbf{u}_{ij}^{\ell} = challenge(C_{ij}^{\ell}, \tau)$ for $\ell \in [r']$, $i \in [n]$, $j \in [n]$. In the rest of the protocol, only $\mathbf{q}_{1ij}^{\ell}, \ldots, \mathbf{q}_{4ij}^{\ell}$, and $(u_{ij}^{\ell})_{|\mathbf{v}|+1}$ will be used.

Each M_k computes $\pi_{ij}^{\ell k} = (\mathbf{p}_{ij}^{\ell k} \| 1 \| \mathbf{v}_{ij}^{\ell k}) = (\mathbf{p}_{ij}^{\ell k} \| 1 \| \mathbf{x}_i^k \| \sum_{j \in \mathcal{R}} \mathbf{r}_{ji}^k \| \mathbf{c}_{1i}^{1k} \| \cdots \| \mathbf{c}_{ni}^{\ell-1,k} \| \mathbf{c}_{ij}^{\ell k})$ for verification, and then computes and publishes the values $\langle \pi_{ij}^{\ell k}, \mathbf{q}_{1ij}^{\ell} \rangle, \ldots, \langle \pi_{ij}^{\ell k}, \mathbf{q}_{4ij}^{\ell} \rangle$. M_i checks these values and complains about M_k that has computed them incorrectly. An uncorrupted M_k may disprove the complaint by publishing the proof and message shares that it received. Due to the linearity of scalar product and the fact that all the vectors have been shared according to the same $(n, t+1)$ Shamir's sharing, if the n scalar product shares correspond to a valid $(n, t+1)$ Shamir's sharing, the shared value is uniquely defined by any $t+1$ shares, and hence, by the shares of some $t+1$ parties that are all from $\mathcal{H}$. Hence, M_i is obliged to use the values it has committed before. The verification phase for C_{ij}^{ℓ} for fixed $\ell \in [r']$, $i \in [n]$, $j \in [n]$ is given in Fig.6. For different C_{ij}^{ℓ}, all the verifications can be done in parallel.

Remaining proof commitment: As the prover, M_i obtains $\mathbf{w}_{ij}^{\ell}$ and $\pi_{ij}^{\ell} = (\mathbf{p}_{ij}^{\ell} \| 1 \| \mathbf{v}_{ij}^{\ell})$ using the algorithms *witness* and *proof*. M_i shares $\mathbf{p}_{ij}^{\ell}$ to $\mathbf{p}_{ij}^{\ell 1}, \ldots, \mathbf{p}_{ij}^{\ell n}$ according to $(n, t+1)$ Shamir's scheme. For each $k \in [n] \setminus \{i\}$, it sends to $\mathcal{F}_{transmit}$ $(\mathsf{transmit}, (\mathsf{p\_share}, \ell, i, j, k), \mathbf{p}_{ij}^{\ell k})$ for M_k.
Challenge generation: Each M_k generates random $\tau_k \leftarrow \mathbb{F}$ and sends to $\mathcal{F}_{transmit}$ the message $(\mathsf{broadcast}, (\mathsf{challenge\_share}, \ell, i, j, k), \tau_k)$. If some party refuses to participate, its share will be omitted. The challenge randomness is $\tau = \tau_1 + \ldots + \tau_n$. Machine M_k generates $\mathbf{q}_{1ij}^{\ell}, \ldots, \mathbf{q}_{4ij}^{\ell}, \mathbf{q}_{5ij}^{\ell}, \mathbf{u}_{ij}^{\ell} = \mathit{challenge}(C_{ij}^{\ell}, \tau)$, then computes $\pi_{ij}^{\ell k} = (\mathbf{p}_{ij}^{\ell k} \| 1 \| \mathbf{v}_{ij}^{\ell k}) = (\mathbf{p}_{ij}^{\ell k} \| 1 \| \mathbf{x}_i^k \| \sum_{j \in \mathcal{R}} \mathbf{r}_{ji}^k \| \mathbf{c}_{1i}^{1k} \| \cdots \| \mathbf{c}_{ni}^{\ell-1,k} \| \mathbf{c}_{ij}^{\ell k})$, and finally computes and broadcasts $\langle \pi_{ij}^{\ell k}, \mathbf{q}_{1ij}^{\ell} \rangle, \ldots, \langle \pi_{ij}^{\ell k}, \mathbf{q}_{4ij}^{\ell} \rangle$.
Scalar product verification: Each M_i verifies the published $\langle \pi_{ij}^{\ell k}, \mathbf{q}_{sij}^{\ell} \rangle$ for $s \in \{1, \ldots, 4\}$. If M_i finds that M_k has computed the scalar products correctly, it sends to $\mathcal{F}_{transmit}$ the message $(\mathsf{broadcast}, (\mathsf{complain}, \ell, i, j, k), 0)$. If some M_k has provided a wrong value, M_i sends to $\mathcal{F}_{transmit}$ $(\mathsf{broadcast}, (\mathsf{complain}, \ell, i, j, k), (1, sh_{sij}^{\ell k}))$, where $sh_{sij}^{\ell k}$ is M_i's own version of $\langle \pi_{ij}^{\ell k}, \mathbf{q}_{sij}^{\ell} \rangle$. Everyone waits for a disproof from M_k. An uncorrupted M_k sends to $\mathcal{F}_{transmit}$ the messages $(\mathsf{publish}, mid)$ for $mid \in \{(\mathsf{x\_share}, i, k), (\mathsf{r\_share}, 1, i, k), \ldots, (\mathsf{r\_share}, |\mathcal{R}|, i, k), (\mathsf{p\_share}, \ell, i, j, k), (\mathsf{c\_share}, 1, 1, i, k), \ldots, (\mathsf{c\_share}, r', n, i, k), (\mathsf{c\_share}, \ell, i, j, k)\}$. Now everyone may construct $\pi_{ij}^{\ell k}$ and verify whether the version provided by M_i or M_k is correct.
Final verification: Given $\langle \pi_{ij}^{\ell k}, \mathbf{q}_{sij}^{\ell} \rangle$ for all $k \in [n]$, $s \in \{1, \ldots, 4\}$, each machine M_v checks if they indeed correspond to valid $(n, t+1)$ Shamir's sharing, and then locally restores $a_{sij}^{\ell} = \langle \pi_{ij}^{\ell}, \mathbf{q}_{sij}^{\ell} \rangle$ for $s \in \{1, \ldots, 4\}$, and checks $(*)$. If the check succeeds, then M_v accepts the proof of M_i for C_{ij}^{ℓ}. Otherwise it immediately sets $mlc_v[i] := 1$.

Figure 6. Verification phase of the real functionality

Finally, each party M_i sends to $\mathcal{Z}$ the message $(\mathsf{blame}, i, \{j \mid mlc_i[j] = 1\})$.

Figure 7. Accusation phase of the real functionality

As described, the probability of cheating successfully in our scheme is proportional to $1/|\mathbb{F}|$. In order to exponentially decrease it, we may run s instances of the verification phase in parallel, as by that time $\mathbf{v}_{ij}^{\ell}$ are already committed. This will not break the HVZK assumption if fresh randomness is used in $\mathbf{p}_{ij}^{\ell}$.

During the message commitment and verification phases, if at any time $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$, the proof for P_j ends with failure, and all uncorrupted machines M_i write $mlc_i[j] := 1$.

Finally, each party outputs the set of parties that it considers malicious. This short phase is given in Fig. 7. The formal UC proof for the real functionality can be found in [30].

5. Efficiency

In this section we estimate the overheads caused by our protocol transformation. The numbers are based on the dominating complexities of the algorithms of linear PCP of [21]. We omit local addition and concatenation of vectors as these are cheap operations.

The preprocessing phase of [21] is done offline and can be re-used, so we will not estimate its complexity here. It can be done with practical overhead [21].

Let n be the number of parties, $t < n/2$ the number of corrupt parties, r the number of rounds, N_g the number of gates, N_w the number of wires, N_x the number of inputs (elements of $\mathbb{F}$), N_r the number of random elements of $\mathbb{F}$, N_c the number of communicated elements of $\mathbb{F}$, and $N_i = N_w - N_x - N_r - N_c$ the number of intermediate wires in the circuit. Then $|\mathbf{v}| = N_x + N_r + N_c$.

Let $\overline{S}(n,k)$ denote the number of field operations used in sharing one field element according to Shamir's scheme with threshold k which is at most nk multiplications. We use $\overline{S^{-1}}(n,k)$ to denote the complexity of verifying if the shares comprise a valid sharing, and of recovering the secret which is also at most nk multiplications. Compared to the original protocol, for each M_i the proposed solution has the following computation/communication overheads.

Initialization: Do Shamir's sharing of one vector of the length N_x in $N_x \cdot \overline{S}(n,t+1)$ field operations. Transmit $t+1$ vectors of the length N_r and one vector of the length N_x to each other party. Do $t+1$ recoverings in $(t+1) \cdot N_r \cdot \overline{S^{-1}}(n,t+1)$. The parties that generate randomness perform $n \cdot N_r \cdot \overline{S}(n,t+1)$ more field operations to compute n more sharings and transmit n more vectors of the length N_r to each other party.

Execution: No computation/communication overheads are present in this phase, except for those caused by the use of the message transmission functionality.

Message commitment: Share all the communication in $rn(n-1) \cdot N_c \cdot \overline{S}(n,t+1)$ operations. Send to each other party rn vectors of the length N_c. Do $r(n-1)$ recoverings in $r(n-1) \cdot N_c \cdot \overline{S^{-1}}(n,t+1)$ operations.

Verification: Compute the proof $\mathbf{p}$ of the length $(4+N_g+N_i)$ in $18N_g + 3 \cdot \overline{FFT}(N_g) + \log N_g + 1$ field operations [21], where $\overline{FFT}(N)$ denotes the complexity of the Fast Fourier Transform, which is $c \cdot N \cdot \log N$ for a small constant c. Share $\mathbf{p}$ in $(4+N_g+N_i) \cdot \overline{S}(n,t+1)$ operations. Send a vector of the length $(4+N_g+N_i)$ to every other party. Broadcast one field element (the τ). Generate the 4 challenges and the state information in $14 \cdot N_g + \log(N_g)$ field operations [21]. Compute and broadcast 4 scalar products of vectors of the length $(5+N_w+N_g)$ (the shares of $\langle (\mathbf{p} \| 1 \| \mathbf{v}), \mathbf{q_s} \rangle$). Compute 4 certain linear combinations of t scalar products and perform 2 multiplications in $\mathbb{F}$ (the products in $a_1 a_2 - a_3 - a_4 u$).

Assuming $N_w \approx 2 \cdot N_g$, for the whole verification phase, this adds up to $\approx rn(2 \cdot \overline{S}(n,t+1)N_g + 3\overline{FFT}(2N_g) + 26nN_g)$ field operations, the transmission of $\approx 4rn^2 N_g$ elements of $\mathbb{F}$, and the broadcast of $4rn^2$ elements of $\mathbb{F}$ per party.

If there are complaints, then at most rn vectors of the length N_c should be published in the message commitment phase, and at most rn vectors of length $(4+N_g+N_i)$ ($\mathbf{p}$ shares), rn^2 vectors of the length N_c (communication shares), $n \cdot (t+1)$ vectors of the length N_r (randomness shares) and n vectors of length N_x (input shares) in the verification phase (per complaining party).

As long as there are no complaints, the only overhead that $\mathcal{F}_{transmit}$ causes is that each message is signed, and each signature is verified.

The knowledge error of the linear PCP of [21] is $\varepsilon = 2N_g/\mathbb{F}$, and the zero-knowledge is δ-statistical for $\delta = N_g/\mathbb{F}$. Hence, the desired error and the circuit size define the field size. If we do not want to use fields that are too large, then the proof can be parallelized as proposed at the end of Sec. 4.

Preprocessing: Parties run the *Randomness generation and commitment* and *Randomness approval* steps of Fig. 3, causing the dealer to learn $r_1,\dots,r_t$. Each r_i is shared as $r_{i1},\dots,r_{in}$ between $P_1,\dots,P_n$.
Sharing: The dealer computes the shares $s_1,\dots,s_n$ of the secret s, using the randomness $r_1,\dots,r_t$ [31], and uses $\mathcal{F}_{transmit}$ to send them to parties $P_1,\dots,P_n$.
Reconstruction: All parties use the publish-functionality of $\mathcal{F}_{transmit}$ to make their shares known to all parties. The parties reconstruct s as in [31].
Verification: The dealer shares each s_i, obtaining $s_{i1},\dots,s_{in}$. It transmits them all to P_i, which verifies that they are a valid sharing of s_i and then forwards each s_{ij} to P_j. *[Message commitment]*
The dealer computes $\mathbf{w} = \mathit{witness}(C,s,r_1,\dots,r_t)$ and $\mathbf{p} = \mathit{proof}(C,(s,r_1,\dots,r_t),\mathbf{w})$. It shares $\mathbf{p}$ as $\mathbf{p}_1,\dots,\mathbf{p}_n$ and transmits $\mathbf{p}_j$ to P_j. *[Proof commitment]*
Each party P_i generates a random $\tau_i \in \mathbb{F}$ and broadcasts it. Let $\tau = \tau_1 + \dots + \tau_n$. Each party constructs $\mathbf{q}_1,\dots,\mathbf{q}_4,\mathbf{q}_5,\mathbf{u} = \mathit{challenge}(C,\tau)$. *[Challenge generation]*
Each party P_i computes $a_{ji} = \langle(\mathbf{p}_i\|1\|s_i\|r_{1i}\|\dots\|r_{ti}\|s_{1i}\|\dots\|s_{ni}),\mathbf{q}_j\rangle$ for $j \in \{1,2,3,4\}$ and broadcasts them. The dealer may complain, in which case $\mathbf{p}_i,s_i,r_{1i},\dots,r_{ti},s_{1i},\dots,s_{ni}$ are made public and all parties repeat the computation of a_{ji}. *[Scalar product verification]*
Each party reconstructs $a_1,\dots,a_4$ and verifies the LPCP equation $(*)$.

Figure 8. LPCP-based verifiable secret sharing

6. Example: Verifiable Shamir's Secret Sharing

In this section we show how our solution can be applied to [31], yielding a verifiable secret sharing (VSS) protocol. Any secret sharing scheme has two phases — sharing and reconstruction — to which the construction presented in this work adds the verification phase.

To apply our construction, we have to define the arithmetic circuits used in [31]. For $i \in \{1,\dots,n\}$ let C_i be a circuit taking $s,r_1,\dots,r_t \in \mathbb{F}$ as inputs and returning $s+\sum_{j=1}^{t} r_j i^j$. If s is the secret to be shared, then C_i is the circuit used by the dealer (who is one of the parties $P_1,\dots,P_n$) to generate the share for the i-th party using the randomness $(r_1,\dots,r_t)$. It computes a linear function, and has no multiplication gates. According to the LPCP construction that we use, each circuit should end with a multiplication. Hence, we append a multiplication gate to it, the other argument of which is 1. Let C be the union of all C_i, it is a circuit with $1+t$ inputs and n outputs.

In the reconstruction phase, the parties just send the shares they have received to each other. A circuit computing the messages of this phase is trivial: it just copies its input to output. We note that $\mathcal{F}_{transmit}$ already provides the necessary publishing functionality for that. Hence, we are not going to blindly follow our verifiable multiparty computation (VMPC) construction, but use this opportunity to optimize the protocol. In effect, this amounts to only verifying the sharing phase of the VSS protocol, and relying on $\mathcal{F}_{transmit}$ to guarantee the proper behavior of parties during the reconstruction. The whole protocol is depicted in Fig. 8.

A few points are noteworthy there. First, the reconstruction and verification phases can take place in any order. In particular, verification could be seen as a part of the sharing, resulting in a 3-round protocol (in the optimistic case). Second, the activities of the dealer in the sharing phase have a dual role in terms of the VMPC construction. They

	Rounds	Sharing	Reconstruction	Verification
Ours	7	$(n-1)\cdot tr$	$n\cdot bc$	$(3n+t+4)(n-1)\cdot tr+5n\cdot bc$
[34]	4	$3n^2\cdot tr$	$O(n^2)\cdot tr$	0
[35]	3	$2n\cdot tr+(n+1)\cdot bc$	$2n\cdot bc$	0
[36]	2	$4n^2\cdot tr+5n^2\cdot bc$	$n^2\cdot bc$	0

Table 1. Comparison of the efficiency of VSS protocols (tr transmissions, bc broadcasts)

form both the *input commitment* step in Fig. 3, as well as the execution step for actual sharing.

Ignoring the randomness generation phase (which takes place offline), the communication complexity of our VSS protocol is the following. In the sharing phase, $(n-1)$ values (elements of $\mathbb{F}$) are transmitted by the dealer, and in the reconstruction phase, each party broadcasts a value. These coincide with the complexity numbers for non-verified secret sharing. In the verification phase, in order to commit to the messages, the dealer transmits a total of $n(n-1)$ values to different parties. The same number of values are forwarded. According to Sec. 5, the proof $\mathbf{p}$ contains $t+n+4$ elements of $\mathbb{F}$. The proof is shared between the parties, causing $(n-1)(t+n+4)$ elements of $\mathbb{F}$ to be transmitted. The rest of the verification phase takes place over the broadcast channel. In the optimistic case, each party broadcasts a value in the challenge generation and four values in the challenge verification phase. Hence, the total cost of the verification phase is $(n-1)(3n+t+4)$ point-to-point transmissions and $5n$ broadcasts of $\mathbb{F}$ elements.

We have evaluated the communication costs in terms of $\mathcal{F}_{transmit}$ invocations, and have avoided estimating the cost of implementing $\mathcal{F}_{transmit}$. This allows us to have more meaningful comparisons with other VSS protocols. We will compare our solution to the 4-round statistical VSS of [34], the 3-round VSS of [35], and the 2-round VSS of [36] (see Table 1). These protocols have different security models and different optimization goals. Therefore, different methods are also selected to secure communication between the parties. Thus, the number of field elements communicated is likely the best indicator of complexity.

The 4*-round statistical VSS of [34].* This information-theoretically secure protocol uses an *information checking protocol* (*ICP*) for transmission, which is a modified version of the *ICP* introduced in [37]. The broadcast channel is also used.

In the protocol, the dealer constructs a symmetric bivariate polynomial $F(x,y)$ with $F(0,0)=s$, and gives $f_i(x)=F(i,x)$ to party P_i. Conflicts are then resolved, leaving honest parties with a polynomial $F^H(x,y)$ that allows the reconstruction of s. The distribution takes $3n^2$ transmissions of field elements using the *ICP* functionality, while the conflict resolution requires $4n^2$ broadcasts (in the optimistic case). The reconstruction phase requires each honest party P_i to send its polynomial f_i to all other parties using the *ICP* functionality, which again takes $O(n^2)$ transmissions.

The 3*-round VSS of [35].* Pedersen's VSS is an example of a computationally secure VSS. The transmission functionality of this protocol is based on homomorphic commitments. Although the goal of commitments is also to ensure message delivery and make further revealing possible, they are much more powerful than $\mathcal{F}_{transmit}$ and *ICP*, so direct comparison is impossible. In the following, let $Comm(m,d)$ denote the commitment of the message m with the witness d. We note that the existence of a suitable *Comm*

is a much stronger computational assumption than the existence of a signature scheme sufficient to implement $\mathcal{F}_{transmit}$.

To share s, the dealer broadcasts a commitment $Comm(s,r)$ for a random r. It shares both s and r, using Shamir's secret sharing with polynomials f and g, respectively. It also broadcasts commitments to the coefficients of f, using the coefficients of g as witnesses. This takes $2n$ transmissions of field elements, and $(n+1)$ broadcasts (in the optimistic case). Due to the homomorphic properties of *Comm*, the correctness of any share can be verified without further communication. The reconstruction requires the shares of s and r to be broadcast, i.e. there are 2 broadcasts from each party.

The 2-round VSS of [36]. This protocol also uses commitments that do not have to be homomorphic. This is still different from $\mathcal{F}_{transmit}$ and *ICP*: commitments can ensure that the same message has been transmitted to distinct parties.

The protocol is again based on the use of a symmetric bivariate polynomial $F(x,y)$ with $F(0,0)=s$ by the dealer. The dealer commits to all values $F(x,y)$, where $1 \le x,y \le n$ and opens the polynomial $F(i,x)$ for the i-th party. The reduction in rounds has been achieved through extra messages committed and sent to the dealer by the receiving parties. These messages can help in conflict resolution. In the optimistic case, the sharing protocol requires $4n^2$ transmissions of field elements and $5n^2$ broadcasts. The reconstruction protocol is similar to [34], with each value of $F(x,y)$ having to be broadcast by one of the parties.

We see that the LPCP-based approach performs reasonably well in verifiable Shamir's sharing. The protocols from the related works have fewer rounds, and the 3-round protocol of [35] clearly also has less communication. However, for a full comparison we have to take into account local computation, as operations on homomorphic commitments are more expensive. Also, the commitments may be based on more stringent computational assumptions than the signature-based communication primitives we are using. We have shown that the LPCP-based approach is at least comparable to similar VSS schemes. Its low usage of the broadcast functionality is definitely of interest.

7. From Finite Fields to Rings

Generalizing a finite field $\mathbb{F}$ to a set of rings (or even to one ring) in a straightforward manner does not work, as we are using Shamir's secret sharing and the LPCP based on finite fields. However, a circuit over rings can be still represented by a circuit over a finite field. We need to add a trunc gate (as defined in Sec. 2) after each gate whose output may become larger than the ring size. The size of $\mathbb{F}$ should be large enough, so that before applying trunc, the output of any gate (assuming that its inputs are truncated to the ring size) would fit into the field. For example, if we want to get a ring of the size 2^n, and we have a multiplication operation, then the field size should be at least 2^{2n}. This, in general, is not the most efficient approach, and we will not explain it in this chapter. The verification of the operations trunc, zext, and bits is similar to the one for rings that we will present.

In this section, we assume that the computation takes place over several rings $\mathbb{Z}_{2^{n_1}}, \ldots, \mathbb{Z}_{2^{n_K}}$. Taking a ring of a size that is not a power of 2 is possible, but less efficient. Instead of Shamir's secret sharing, we now have to use *additive secret sharing* (see Chapter 1 for details). Each value is shared in the corresponding ring in which it is used.

As additive secret sharing does not support a threshold, the prover has to repeat the proof with each subset of t verifiers separately (excluding the sets containing the prover itself). The proof succeeds if and only if the outcomes of all the verifier sets are satisfiable. The number of verification sets is exponential in the number of parties, but it can be reasonable for a small number of parties.

7.1. Additional Operations for Rings

We can now define the verification for the remaining gate operations defined in Sec. 2 that we could not verify straightforwardly in $\mathbb{F}$. If we need to compute $z := \mathsf{trunc}(x)$, we *locally* convert the shares over the larger ring to shares over the smaller ring, which is correct as the sizes of the rings are powers of 2, and so the size of the smaller ring divides the size of the larger ring. However, if we need to compute $z := \mathsf{zext}(x)$, then we cannot just covert the shares of committed z locally, as zext is not an inverse of trunc, and we need to ensure that all the excessive bits of z are 0.

Formally, the gate operations of Sec. 2 are verified as follows:

1. The bit decomposition operation $(z_0, \ldots, z_{n-1}) := \mathsf{bits}(z)$:
   check $z = z_0 + z_1 \cdot 2 + \cdots + z_{n-1} 2^{n-1}$;
   check $\forall j : z_j \in \{0,1\}$.
2. The transition from $\mathbb{Z}_{2^m}$ to a smaller ring $\mathbb{Z}_{2^n}$: $z := \mathsf{trunc}(x)$:
   compute locally the shares of z from x, do not perform any checks.
3. The transition from $\mathbb{Z}_{2^n}$ to a larger ring $\mathbb{Z}_{2^m}$: $z := \mathsf{zext}(x)$:
   compute locally the shares of $y := \mathsf{trunc}(z)$ from z;
   check $x = y$;
   check $z = z_0 + z_1 \cdot 2 + \ldots + z_{n-1} \cdot 2^{n-1}$;
   check $\forall j : \ z_j \in \{0,1\}$.

As the computation takes place over a ring, we can no longer apply the LPCP used in Sec. 3. In Sec. 7.2, we propose some other means for making the number of rounds in the verification phase constant.

7.2. Pushing More into the Preprocessing Phase

A significant drawback of the construction presented in Sec. 4 is that the local computation of the prover is superlinear in the size of the circuit ($|C| \log |C|$). Now we introduce a slightly different setting that requires a more expensive offline precomputation phase, but makes the verification more efficient. The main idea is that if the circuit does not contain any multiplication gates, then linear secret sharing allows the verifiers to repeat the entire computation of the prover locally, getting the shares of all the outputs in the end. For an arbitrary circuit, we may get rid of the multiplications using Beaver triples.

Consider a circuit C_{ij}^{ℓ} being verified. For each multiplication gate, a Beaver triple is generated in the corresponding ring $\mathbb{Z}_{2^n}$. The triple is known by the prover, and it is used only in the verification, but not in the computation itself. The triple generation is performed using an ideal functionality $\mathcal{F}_{Bt}$ (see Fig. 9) that generates Beaver triples and shares them amongst the parties. Additionally, this functionality generates and shares random bits, which will be used similarly to Beaver triples: at some moment, b' is published, so that $b = (b' + r_b) \mod 2$. These random bits are not used in multiplication,

$\mathcal{F}_{Bt}$ works with unique wire identifiers id, encoding a ring size $n(id)$ of the value of this wire. It stores an array $mult$ of the shares of Beaver triples for multiplication gates, referenced by unique identifiers id, where id corresponds to the output wire of the corresponding multiplication gate. It also stores an independent array bit, referenced by id, that stores the shares of random bit vectors that will be used in the bit decomposition of the wire identified by id.

Initialization: On input (init) from the environment, set $mult := []$, $bit := []$.

Beaver triple distribution: On input (beaver, j, id) from M_i, check if $mult[id]$ exists. If it does, take $(r_x^1,\ldots,r_x^n,r_y^1,\ldots,r_y^n,r_{xy}^1,\ldots,r_{xy}^n) := mult[id]$. Otherwise, generate $r_x \xleftarrow{\$} \mathbb{Z}_{n(id)}$ and $r_y \xleftarrow{\$} \mathbb{Z}_{n(id)}$. Compute $r_{xy} = r_x \cdot r_y$. Share r_x to r_x^k, r_y to r_y^k, r_{xy} to r_{xy}^k. Assign $mult[id] := (r_x^1,\ldots,r_x^n,r_y^1,\ldots,r_y^n,r_{xy}^1,\ldots,r_{xy}^n)$. If $j \neq i$, send r_x^i, r_y^i, r_{xy}^i to M_i. Otherwise, send $(r_x^1,\ldots,r_x^n,r_y^1,\ldots,r_y^n,r_{xy}^1,\ldots,r_{xy}^n)$ to M_i.

Random bit distribution: On input (bit, j, id) from M_i, check if $bit[id]$ exists. If it does, take $(\mathbf{b}^1,\ldots,\mathbf{b}^n) := bit[id]$. Otherwise, generate a bit vector $\mathbf{b} \xleftarrow{\$} (\mathbb{Z}_2)^{n(id)}$ and share it to $\mathbf{b}^k$. Assign $bit[id] := (\mathbf{b}^1,\ldots,\mathbf{b}^n)$. If $j \neq i$, send $\mathbf{b}^i$ to M_i. Otherwise, send $(\mathbf{b}^1,\ldots,\mathbf{b}^n)$ to M_i.

Figure 9. Ideal functionality $\mathcal{F}_{Bt}$

and they are used to ensure that b is a bit. Namely, if $b' = 0$, then $b = r_b$, and $b = 1 - r_b$ otherwise. If r_b is indeed a bit (which can be proved in the preprocessing phase), then b is also a bit.

7.3. Modified Real Functionality

Due to additional preprocessing, the real functionality becomes somewhat different from the real functionality of Sec. 4.

Preprocessing. This is a completely offline preprocessing phase that can be performed before any inputs are known. The following values are precomputed for the prover M_i:

- Let id be the identifier of a circuit wire that needs a proof of correctness of its bit decomposition (proving that $z_j \in \{0,1\}$ and $z = z_0 + z_1 \cdot 2 + \cdots + z_{n(id)-1} \cdot 2^{n(id)-1}$). Each party M_k sends query (bit, i, id) to $\mathcal{F}_{Bt}$. The prover M_i receives all the shares $(\mathbf{b}^1,\ldots,\mathbf{b}^n)$, and each verifier just the share $\mathbf{b}^k$. Let $\bar{\mathbf{b}}_i^k$ be the vector of all such bit shares of the prover M_i issued to M_k.
- Let id be the identifier of a multiplication gate of M_i, where both inputs are private. Each party M_k sends a query (beaver, i, id) to $\mathcal{F}_{Bt}$. The prover M_i receives all the shares $(r_x^1,\ldots,r_x^n,r_y^1,\ldots,r_y^n,r_{xy}^1,\ldots,r_{xy}^n)$, and each verifier just the shares (r_x^k, r_y^k, r_{xy}^k).

Initialization. The same as in Sec. 4. The inputs $\mathbf{x}_i$ and the circuit randomness $\mathbf{r}_i$ are shared.

Message commitment. The first part of this phase is similar to Sec. 4. In the message commitment phase, all the n parties finally commit their sent messages $\mathbf{c}_{ij}^\ell$ for each round $\ell \in [r']$ by sharing them to $\mathbf{c}_{ij}^{\ell k}$ and sending these shares to the other parties. This phase is given in Fig. 10. Let $\mathbf{v}_{ij}^\ell = (\mathbf{x}_i \| \mathbf{r}_i \| \mathbf{c}_{1i}^1 \| \cdots \| \mathbf{c}_{ni}^{\ell-1} \| \mathbf{c}_{ij}^\ell)$ be the vector of inputs and outputs

Message sharing: As a sender, M_i shares $\mathbf{c}_{ij}^{\ell}$ to $\mathbf{c}_{ij}^{\ell k}$ according to Shamir's secret sharing scheme. For each $k \in [n]$, M_i sends to $\mathcal{F}_{transmit}$ the messages $(\mathsf{transmit}, (\mathsf{c\_share}, \ell, i, j, k), \mathbf{c}_{ij}^{\ell k})$ for M_j.

Public values: The prover M_i constructs the vector $\mathbf{b}_i^{\ell}$ which denotes which entries of communicated values of $\bar{\mathbf{b}}_i^{\ell}$ (related to communication values) should be flipped. Let $\mathbf{p}_i^{\ell}$ be the vector of the published values c' so that $c = (c' + r_c)$ is a masqued communication value. M_i sends to $\mathcal{F}_{transmit}$ a message $(\mathsf{broadcast}, (\mathsf{communication\_public}, \ell, i), (\mathbf{p}_i^{\ell}, \mathbf{b}_i^{\ell}))$.

Message commitment: Upon receiving $((\mathsf{c\_share}, \ell, i, j, k), \mathbf{c}_{ij}^{\ell k})$ and $(\mathsf{broadcast}, (\mathsf{communication\_public}, \ell, i), (\mathbf{p}_i^{\ell}, \mathbf{b}_i^{\ell}))$ from $\mathcal{F}_{transmit}$ for all $k \in [n]$, the machine M_j checks if the shares correspond to $\mathbf{c}_{ij}^{\ell}$ it has already received. If only $c_{ij}^{\ell s\prime}$ is published for some $c_{ij}^{\ell s}$, then it checks $c_{ij}^{\ell s} = c_{ij}^{\ell s\prime} + r_c$ for the corresponding preshared randomness r_c (related to the Beaver triple). If something is wrong, M_j sends $(\mathsf{publish}, (\mathsf{message}, \ell, i, j))$ to $\mathcal{F}_{transmit}$, so now everyone sees the values that it has actually received from M_i, and each (uncorrupted) M_k should now use $\mathbf{c}_{ij}^{\ell k} := \mathbf{c}_{ij}^{\ell}$. If the check succeeds, then M_i sends to $\mathcal{F}_{transmit}$ $(\mathsf{forward}, (\mathsf{c\_share}, \ell, i, j, k))$ for M_k for all $k \in [n] \setminus \{i\}$.

Figure 10. Message commitment phase of the real functionality

to the circuit C_{ij}^{ℓ} that M_i uses to compute the ℓ-th message to M_j. After this phase, M_i has shared $\mathbf{v}_{ij}^{\ell}$ among all n parties. Let $\mathbf{v}_{ij}^{\ell k}$ be the share of $\mathbf{v}_{ij}^{\ell}$ given to machine M_k.

The proving party now also publishes all the public Beaver triple communication values: for each $c = (c' + r_c)$, it publishes c'. It also publishes a bit b'_{ij} for each communicated bit b_{ij} that requires proof of being a bit. For the communicated values of $\mathbf{c}_{ij}^{\ell}$, publishing only the value $\mathbf{c'}_{ij}^{\ell}$ is sufficient, and $\mathbf{c}_{ij}^{\ell}$ itself does not have to be reshared.

During the message commitment phase, if at any time $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{transmit}$, the proof for P_j ends with failure, and all uncorrupted machines M_i write $mlc_i[j] := 1$.

Verification phase. The proving party publishes all the remaining public Beaver triple values, and all the remaining bits b'_{ij} for each bit b_{ij} that require proof of being a bit (see Fig. 11). For each operation where $z \in \mathbb{Z}_{2^{n_e}}$, the prover commits by sharing the value of z in the ring $\mathbb{Z}_{2^{n_e}}$.

After all the values are committed and published, each verifier M_k does the following locally:

- Let $\bar{\mathbf{b}}_i^k$ be the vector of precomputed random bit shares for the prover M_i, and $\mathbf{b}_i$ the vector of published bits. For each entry $\bar{b}_{ij}^k$ of $\bar{\mathbf{b}}_{\mathbf{i}}^{\mathbf{k}}$, if $b_{ij} = 1$, then the verifier takes $1 - \bar{b}_{ij}^k$, and if $b_{ij} = 1$, then it takes $\bar{b}_{ij}^k$ straightforwardly. These values will now be used in place of all shares of corresponding bits.
- For all Beaver triple shares (r_x^k, r_y^k, r_{xy}^k) of M_i, the products $x' r_y^k$, $y' r_x^k$, and $x'y'$ are computed locally.

As a verifier, each M_k computes each circuit of the prover on its local shares. Due to preshared Beaver triples, the computation of $+$ and $*$ gates is linear, and hence, communication between the verifiers is not needed.

The correctness of operations $(z_1, \ldots, z_{n_e}) := \mathsf{bits}(z)$, $z = \mathsf{zext}(x)$, and $z = \mathsf{trunc}(x)$ is verified as shown in Sec. 7.1. The condition $\forall j : \ z_j \in \{0, 1\}$ can be ensured as follows:

Remaining proof commitment: The prover M_i constructs the vector $\mathbf{b}_i^\ell$ that denotes which entries of non-communicated values $\bar{\mathbf{b}}_i^\ell$ should be flipped. Let $\mathbf{p}_i^\ell$ be the vector of the published values z' so that $z = (z' + r_z)$ is a masqued non-communicated value. M_i sends to $\mathcal{F}_{transmit}$ a message $(\mathsf{broadcast}, (\mathsf{remaining\_public}, \ell, i), (\mathbf{p}_i^\ell, \mathbf{b}_i^\ell))$.
For each operation $z = \mathsf{zext}(x)$, $z = \mathsf{trunc}(x)$, M_i shares z to z^k. Let $\mathbf{z}_i^{\ell k}$ be the vector of all such shares in all the circuits of M_i. It sends $((\mathsf{z\_share}, \ell, i, k), \mathbf{z}_i^{\ell k})$ to $\mathcal{F}_{transmit}$.
Local computation: After receiving all the messages $(\mathsf{broadcast}, (\mathsf{remaining\_public}, \ell, i), (\mathbf{p}_i^\ell, \mathbf{b}_i^\ell))$ and $((\mathsf{z\_share}, \ell, i, k), \mathbf{z}_i^{\ell k})$, each verifying party M_k locally computes the circuits of the proving party M_i on its local shares, collecting the necessary linear equality check shares. In the end, it obtains a set of shares $A_1\mathbf{x}_1^k, \ldots, A_K\mathbf{x}_K^k$. M_i computes and publishes $\mathbf{d}_{ij}^{\ell k} = (A_1\mathbf{x}_1^k \| \ldots \| A_K\mathbf{x}_K^k)$.
Complaints and final verification: The prover M_i knows how a correct verification should proceed and hence, it may compute the values $\mathbf{d}_{ij}^{\ell k}$ itself. If the published $\mathbf{d}_{ij}^{\ell k}$ is wrong, then the prover accuses M_k and publishes all the shares sent to M_k using $\mathcal{F}_{transmit}$. All the honest parties may now repeat the computation on these shares and compare the result. If the shares $\mathbf{d}_{ij}^{\ell k}$ correspond to $\mathbf{0}$, then the proof of M_i for C_{ij}^ℓ is accepted. Otherwise, each honest party now immediately sets $mlc_v[i] := 1$.

Figure 11. Verification phase of the real functionality

using the bit r_{z_j} shared in the preprocessing phase, and z'_j published in the commitment phase, each party locally computes the share of $z_j \in \{0,1\}$ as r_{z_j} if $z'_j = 0$, and $1 - r_{z_j}$ if $z'_j = 1$. In the case of zext, the verifiers compute the shares of y locally, and take the shares of z that are committed by the prover in the commitment phase. Now, the checks of the form $x - y = 0$ and $z_0 + z_1 \cdot 2 + \ldots + z_{n_e-1} \cdot 2^{n_e-1} - z = 0$ are left. Such checks are just linear combinations of the shared values. As the parties cannot verify locally if the shared value is 0, they postpone these checks to the last round.

For each multiplication input, the verifiers need to check $x = (x' + r_x)$, where x is either the initial commitment of x, or the value whose share the verifier has computed locally. The shares of $x' + r_x$ can be different from the shares of x, and that is why an online check is not sufficient. As $z = x * y = (x' + r_x)(y' + r_y) = x'y' + x'r_y + y'r_x + r_{xy}$, the verifiers compute locally $z^k = x'y' + x'r_y^k + y'r_x^k + r_{xy}^k$ and proceed with z^k. The checks $x = (x' + r_x)$ and $y = (y' + r_y)$ are delayed.

Finally, the verifiers come up with the shares $\bar{c}_{ij}^{\ell k}$ of the values $\bar{c}_{ij}^\ell$ that should be the outputs of the circuits. The verifiers have to check $c_{ij}^\ell = \bar{c}_{ij}^\ell$, but the shares $c_{ij}^{\ell k}$ and $\bar{c}_{ij}^{\ell k}$ can be different. Again, an online linear equality check is needed for each $c_{ij}^{\ell k}$.

In the end, the verifiers get the linear combination systems $A_1\mathbf{x}_1 = 0, \ldots, A_K\mathbf{x}_K = 0$, where $A_i\mathbf{x}_i = 0$ has to be checked in $\mathbb{Z}_{2^{n_i}}$. They compute the shares of $\mathbf{d}_i^k := A_i\mathbf{x}_i^k$ locally. If the prover is honest, then the vectors $\mathbf{d}_i^k$ are just shares of a zero vector and hence, can be revealed without leaking any information.

Unfortunately, in a ring we cannot merge the checks $\mathbf{d}_i = \mathbf{0}$ into one $\langle \mathbf{d}_i, \mathbf{s}_i \rangle = 0$ due to a large number of zero divisors (the probability of cheating becomes too high). However, if the total number of parties is 3, then there are 2 verifiers in a verifying set. They want to check if $\mathbf{0} = \mathbf{d}_i = \mathbf{d}_i^1 + \mathbf{d}_i^2$, which is equivalent to checking whether $\mathbf{d}_i^1 = -\mathbf{d}_i^2$. For this, take a collision-resistant hash function and publish $h_{ij}^{\ell 1} := h((\mathbf{d}_1^1 \| \ldots \| \mathbf{d}_K^1))$ and $h_{ij}^{\ell 2} := h(-(\mathbf{d}_1^2 \| \ldots \| \mathbf{d}_K^2))$. Check $h_{ij}^{\ell 1} = h_{ij}^{\ell 2}$.

Accusation. The accusation phase is the same as in Sec. 4.

The formal security proofs can be found in [38].

8. Conclusions and Further Work

We have proposed a scheme transforming passively secure protocols to covertly secure ones, where a malicious party can skip detection only with negligible probability. The protocol transformation proposed here is particularly attractive to be implemented on top of some existing, highly efficient, passively secure SMC framework. The framework will retain its efficiency, as the time from starting a computation to obtaining the result at the end of the execution phase will not increase. Also, the overheads of verification, which are proportional to the number of parties, will be rather small due to the small number of *computing parties* in all typical SMC deployments (the number of *input* and *result parties* (see Chapter 1 for details) may be large, but they can be handled separately).

The implementation would allow us to study certain trade-offs. Sec. 5 shows that the proof generation is still slightly superlinear in the size of circuits, due to the complexity of FFT. Shamir's secret sharing would allow the parties to commit to some intermediate values in their circuits, thereby replacing a single circuit with several smaller ones, and decreasing the computation time at the expense of communication. The usefulness of such modifications and the best choice of intermediate values to be committed would probably largely depend on the actual circuits.

Note that the verifications could be done after each round. This would give us security against active adversaries quite cheaply, but would incur the overhead of the verification phase during the runtime of the actual protocol.

References

[1] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.

[2] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[3] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

[4] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[5] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[6] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[7] Octavian Catrina and Sebastiaan de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2010.

[8] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.

[9] Matthew K. Franklin, Mark Gondree, and Payman Mohassel. Communication-efficient private protocols for longest common subsequence. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *Lecture Notes in Computer Science*, pages 265–278. Springer, 2009.

[10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[11] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[12] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *Twenty-Second Annual IEEE Conference on Computational Complexity, CCC'07.*, pages 278–291. IEEE, 2007.

[13] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In Daniele Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[14] Sanjeev Arora and Shmuel Safra. Probabilistic Checking of Proofs: A New Characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[15] Silvio Micali. CS Proofs (Extended Abstract). In *FOCS*, pages 436–453. IEEE Computer Society, 1994.

[16] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, New York, NY, USA, 1992. ACM.

[17] A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[18] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[19] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[20] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. *IACR Cryptology ePrint Archive*, 2013:121, 2013.

[21] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[22] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

[23] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, 2012.

[24] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[25] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[26] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2014.

[27] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[28] Ralph C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, 1979.

[29] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch verification of short signatures. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.

[30] Peeter Laud and Alisa Pankova. Verifiable Computation in Multiparty Protocols with Honest Majority. Cryptology ePrint Archive, Report 2014/060, 2014. `http://eprint.iacr.org/`.

[31] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[32] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Feigenbaum [39], pages 420–432.

[33] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From Input Private to Universally Composable Secure Multi-party Computation Primitives. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium*. IEEE, 2014.

[34] Ranjit Kumaresan, Arpita Patra, and C. Pandu Rangan. The round complexity of verifiable secret sharing: The statistical case. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 431–447. Springer, 2010.

[35] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Feigenbaum [39], pages 129–140.

[36] Michael Backes, Aniket Kate, and Arpita Patra. Computational verifiable secret sharing revisited. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 590–609. Springer, 2011.

[37] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *STOC*, pages 73–85. ACM, 1989.

[38] Peeter Laud and Alisa Pankova. Precomputed verification of multiparty protocols with honest majority. In preparation, 2015.

[39] Joan Feigenbaum, editor. *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*. Springer, 1992.