Preprocessing-Based Verification of Multiparty Protocols with Honest Majority

Roman Jagomgis, Peeter Laud, Alisa Pankova
Cybernetica AS, Estonia

Abstract

This paper presents a generic "GMW-style" method for turning passively secure protocols into protocols secure against covert attacks, adding relatively cheap offline preprocessing and post-execution verification phases. The execution phase, after which the computed result is already available to the parties, has only negligible overhead.

Our method uses shared verification based on precomputed and -verified multiplication triples. The verification proceeds by the verifiers repeating the computations of the prover in secret-shared manner, checking that they obtain the same messages that the prover sent out during execution. The verification preserves the privacy guarantees of the original protocol. It is applicable to protocols doing computations over finite rings, even if the same protocol performs its computation over several distinct rings at once. We apply our verification method to the Sharemind platform for secure multiparty computations (SMC), evaluate its performance and compare it to other existing SMC platforms offering security against stronger than passive attackers.

1 Introduction

Suppose that mutually distrustful parties communicating over a network want to solve a common computational problem. It is known that such a computation can be performed in a manner that the participants only learn their own outputs and nothing else [35], regardless of the functionality that the parties actually compute. This general result is based on a construction expensive in both computation and communication, but now there exist more efficient general secure multiparty computation (SMC) platforms [11, 15, 21, 25], as well as various protocols optimized to solve concrete problems [14, 18, 20, 31].

Two main kinds of adversaries against SMC protocols are typically considered: passive and active. The highest performance and greatest variety is achieved for protocols secure against passive adversaries. In practice one would like to achieve stronger security guarantees. Achieving security against active adversaries may be expensive, hence intermediate adversary classes (between passive and active) have been introduced.

It is often a sufficient deterrent if an active adversary is going to be eventually detected. It does not have to happen immediately after the malicious act. Hence ideas from verifiable computation (VC) [33] are applicable to SMC. In general, VC allows a weak client to outsource a computation to a more powerful server that accompanies the computed result with a proof of correct computation that is relatively easy for the weak client to verify. Similar ideas can be used to strengthen protocols secure against passive adversaries: after execution, each party will prove to others that it has correctly followed the protocol.

In this work we propose a distributed verification mechanism allowing one party (the prover) to convince others (the verifiers) that it followed the protocol correctly. In our mechanism, the intermediate computation values of the prover are secret-shared (using a linear secret-sharing scheme) amongst a set of verifier parties where at least one verifier is honest. The verifiers repeat the prover's computations, using verifiable hints from the prover. The verification is zero-knowledge to any minority coalition of parties.

Prover's hints are based on precomputed multiplication triples [6] (*Beaver triples*), which we adapt for verification. Before starting the verification (and even the execution), the verifiers hold shares of sufficiently many triples. Importantly, at this time, they are already convinced in the correctness of the triples. During verification, the correctness of precomputed triples implies the correctness of prover's computations.

The entire construction constitutes a variant of the GMW compiler [17,35] from passively to actively secure protocols, showing that this technique can be highly effective. Our verification phase can be seen as an interactive proof, where the prover uses correlated randomness to make the proof, and the verifier has been implemented using SMC to ensure its correct behaviour and prover's privacy.

Applying this verification mechanism n times to any n-party computation protocol, with each party acting as the prover in one instance, gives us a protocol secure against covert (if verification is performed at the end) or fully malicious (if each protocol round is immediately verified) adversaries corrupting a minority of parties. In this work we apply that mechanism to the SMC protocol set [11] employed in the Sharemind platform [10], demonstrating for the first time a method to achieve security against active adversaries for this efficient [39] protocol set, the deployments [12, 37, 60] of which include the largest SMC applications ever [7, 8]. We discuss the difficulties with previous methods in Sec. 2.

Cost of precomputation. There exist other protocols for SMC secure against active adversaries (see Sec. 2) with very modest overheads due to the need to verify the behaviour of parties. Our verification phase, even while having a reasonable cost of its own, is not competitive with these approaches. However, the efficiency of the verification comes at the expense of very costly precomputation (see Sec. 2), significantly hampering the deployment. Our approach also has the precomputation phase, which is still the most expensive part of the protocol, but it is orders of magnitude faster than previous methods (see Sec. 6) and may actually serve as a partial replacement for them (see Sec. 7).

The reduction of the total cost of actively secure computation is the main benefit of our work. We achieve this through novel constructions of verifiable computing, reducing the correctness of computations to the correctness of pre-generated multiplication triples and tuples for other operations.

2 Related Work

Several techniques exist for two-party or multiparty computation secure against malicious adversaries. We are aware of implementations based on garbled circuits [41, 49], on additive sharing with MACs to check for correct behaviour [24–26], on Shamir's secret sharing [21, 58], and on the GMW protocol [35] paired with actively secure oblivious transfer [49]. Different techniques suit the secure execution of different kinds of computations, as we discuss below. The verification technique we propose in this paper is mostly suitable for secret-sharing based SMC, with no preference towards the algebraic structures underlying the computation.

Our protocol uses precomputed multiplication triples, and also precomputed tuples for other operations to verify whether parties have followed the protocol. Such triples [6] are used by several existing SMC frameworks, including SPDZ [25] or ABY [30]. Differing from them, we use the triples not for performing computations, but for verifying them. This is a new idea that allows us to sidestep the most significant difficulties in pre-generating the tuples.

The difficulty is, that the precomputed tuples for secure computation must be private. Heavyweight cryptographic tools are used to generate them under the same privacy constraints as obeyed by the main phase of the protocol. Existing frameworks utilize homomorphic [30, 51, 53] or somewhat (fully) homomorphic encryption systems [13,24] or oblivious transfer [49]. For ensuring the correctness of tuples, the generation is followed by a much cheaper correctness check [24]. Our approach keeps the correctness check, but the generation can be done "in the open" by the party whose behaviour is going to be checked.

Previously, methods for post-execution verification of the correct behaviour of protocol participants have been presented in [22, 43]. Laud and Pankova's method [43] is similar to our work in their applicability (it is in principle possible to verify any computation), as well as in the provided guarantees (a malicious party has only a negligible chance of going undetected). Their approach still has significant computational costs during the verification phase that is run after the execution of the protocol. Damgård et al's method [22] is more specific. It applies specifically to SMC protocols, and only to protocols with certain structure, which mildly restricts its applicability. Both works are instances of a more generic *verifiable computation* problem, which may be approached in a number of different ways [1, 33, 36, 40, 48]. In the setting of multiparty computation, the verifiers may also differ from the protocol participants [2, 4]. We note that the general outline of our verification scheme is similar to [4] — we both commit to certain values during protocol execution and perform computations with them afterwards. However, the commited values and the underlying commitment scheme are very different. One important resulting difference is that our work can be straightforwardly applied to computation over rings.

We apply our verification to the protocol set of Sharemind [10], which is based on additive sharing over finite rings (typically: integers of certain bit-width) among three *computing* parties (the number of parties providing inputs or receiving outputs may be much larger). The protocol set tolerates one passive corruption. Existing MAC-based methods for ensuring the correct behaviour of parties are not applicable to this protocol set, because these methods presume the sharing to be done over a finite field. Also, these methods can protect only a limited set of operations that the computing parties may do, namely the linear combinations and declassification. Sharemind derives its efficiency from the great variety of protocols it has and from the various operations that may be performed with the shares.

The verification step converts a protocol secure against a passive adversary to a protocol secure against *covert* adversary [3] that is prevented from deviating from the prescribed protocol by a non-negligible chance of getting caught. In our case, the probability of not being caught is negligible, based on the properties of underlying message transmission functionality (signatures), hash functions, and the protocols that generate offline preshared randomness. When applied to Sharemind, we obtain an efficient protocol set for three computing parties, which can tolerate one actively corrupted party.

We believe that in most situations, where sufficiently strong legal or contractual frameworks are in place, providing protection against covert adversaries is sufficient to cover possible active corruptions. The computing parties should have a contract describing their duties in place anyway [28], this contract can also specify appropriate punishments for being caught deviating from the protocol.

Complexity of actively secure integer multiplication

We are interested in bringing security against active adversaries to integer and floating-point operations, to be used in secure statistical analyses [8], scientific computations [37] or risk analysis [7]. Such applications use protocols for different operations on private data, but an important subprotocol in all of them is the multiplication of private integers. Hence, let us study the state of the art in performing integer multiplications in actively secure computation protocol sets. All times reported below are amortized over the parallel execution of many protocol instances. All reported tests have used modern (at the time of the test) servers (one per party), connected to each other over a local-area network.

Such protocol sets are based either on garbled circuits or secret sharing (over various fields). Lindell and Riva [46] have recently measured the performance of maliciously secure garbled circuits using state-of-the-art optimizations. They have found that the total execution time for a single AES circuit is around 80ms, when doing 1024 executions in parallel and using the security parameter $\eta = 80$. The size of their AES circuit is 6800 non-XOR gates. According to [29], a 32-bit multiplier can be built with ca. 1700 non-XOR gates. Hence we extrapolate that such multiplication may take ca. 20ms under the same conditions. Our extrapolation cannot be very precise due to the very different shape of the circuits computing AES or multiplication, but it should be valid at least as an order-of-magnitude approximation.

A protocol based on secret sharing over $\mathbb{Z}_2$ [49] would use the same circuit to perform integer multiplication. In [32], a single non-XOR gate is estimated to require ca. 70µs during preprocessing (with two parties). Hence a whole 32-bit multiplier would require ca. 120ms. As the preprocessing takes the lion's share of total costs, there is no need for us to estimate the performance of the online phase.

Recent estimations of the costs of somewhat homomorphic encryption based preprocessing for maliciously secure multiparty computation protocols based on additively secret sharing over $\mathbb{Z}_p$ are hard to come by. In [23], the time to produce a multiplication triple for $p \approx 2^{64}$ is estimated as 2ms for covert security and 6ms for fully malicious security (with two parties, with $\eta = 40$). We presume that the cost is smaller for smaller p, but for $p \approx 2^{32}$, it should not be more than twice as fast. On the other hand, the increase of η to 80 would double the costs [23]. In [24], the time to produce a multiplication triple for $p \approx 2^{32}$ is measured to be 1.4ms (two parties, $\eta = 40$, escape probability of a cheating adversary bounded by 20%).

The running time for actively secure multiplication protocol for 32-bit numbers shared using Shamir's sharing has been reported as 9ms in [21] (with four parties, tolerating a single malicious party). We are not aware of any more modern investigations into Shamir's secret sharing based SMC.

A more efficient N-bit multiplication circuit is proposed in [27], making use computations in $\mathbb{Z}_2$ and in $\mathbb{Z}_p$ for $p \approx N$. Using this circuit instead of the one reported in [29] might improve the running times of certain integer multiplication protocols. But it is unclear, what is the cost of obtaining multiplication triples for $\mathbb{Z}_p$.

In this work, we present a set of protocols that is capable of performing a 32-bit integer multiplication with covert security (on a LAN, with three parties, tolerating a single actively corrupted party, $\eta = 80$, negligible escape probability for a cheating adversary) in 14µs. This is around two orders of magnitude faster than the performance reported above. In practice, we may achieve even better performance by making use of covertness and not verifying all protocol runs. See Sec. 7 for discussion.

In concurrent work [38], the oblivious transfer methods of [32] have been extended to construct SPDZ multiplication triples over $\mathbb{Z}_p$. They report amortized timings of ca. 200µs for a single triple with two parties on a 1Gbps network, where $p \approx 2^{128}$ and $\eta = 64$. Reducing the size of integers would probably also reduce the timings, perhaps even bringing them to the same order of magnitude with our results. But their techniques (as well as most others described here) only work for finite fields, not rings. For fields, there exist methods to reduce the number of discarded triples during triple verification, which also apply for us. See Sec. 5.3 for discussion.

3 Ideal Functionality

Notation. Throughout this work, we use $\vec{x}$ to denote vectors, where x_i is the i-th coordinate of $\vec{x}$. All operations on vectors are defined elementwise. We denote $[n] = \{1, \ldots, n\}$.

Circuits. An *arithmetic circuit* over rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$ consists of gates performing arithmetic operations, and connections between them. An operation may be either an addition, constant multiplication, or multiplication in one of the rings $\mathbb{Z}_{n_k}$. It may also be "$x = \mathsf{trunc}(y)$" or "$y = \mathsf{zext}(x)$" for $x \in \mathbb{Z}_{n_x}$ and $y \in \mathbb{Z}_{n_y}$, where $n_x < n_y$. The first of them computes $x = y \bmod n_x$, while the second lifts $x \in \mathbb{Z}_{n_x}$ to the larger ring $\mathbb{Z}_{n_y}$. Finally, there is an operation $(z_1, \ldots, z_{\lceil \log n_x \rceil}) = \mathsf{bits}(x)$ that decomposes $x \in \mathbb{Z}_{n_x}$ into bits. This operation could be implemented through other listed operations, but it occurs so often in our protocols, and can be verified much more efficiently, so it makes sense to consider it separately.

This set of gates is sufficient to represent any computation; any gates with other operations can be expressed as a composition of the available ones. Nevertheless, the verifications designed for special gates may be more efficient. In the protocol set of Sharemind [11], the parties do employ some other operations for computing the outgoing messages; we can handle all of them.

Execution Functionality. We specify our verifiable execution functionality in the universal composability (UC) framework [16]. Such specification allows us to precisely state the security properties of the execution.

We have n parties (indexed by $[n]$), where $\mathcal{C} \subseteq [n]$ are corrupted for $|\mathcal{C}| < n/2$ (we denote $\mathcal{H} = [n] \backslash \mathcal{C}$). There is a secure channel between each pair of parties. The protocol is synchronous; it 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 a publicly known arithmetic circuit C_{ij}^{ℓ} over rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$. The honest parties are using these circuits to compute their outgoing messages, while the corrupted parties can send anything.

The circuit C_{ij}^{ℓ} computes the ℓ-th round messages $\vec{m}_{ij}^{\ell}$ to the party $j \in [n]$ from the input $\vec{x}_i$, randomness $\vec{r}_i$ and the messages $\vec{m}_{j'i}^{k}$ ($k < \ell$) that P_i has received before. All values $\vec{x}_i, \vec{r}_i, \vec{m}_{ij}^{\ell}$ are vectors over rings $\mathbb{Z}_N$. 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 $\mathcal{Z}$ (specifying the computations of all parties in the form of circuits and the inputs of honest parties), as well as the adversary $\mathcal{A}_S$, is given on Fig. 1.

In addition to the computation results, $\mathcal{F}_{vmpc}$ outputs to each party a set $\mathcal{M}$ of parties deviating 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.

4 The Real Protocol

4.1 High-level Overview

Before going to the details, let us give a general look of transforming a protocol, defined by circuits C_{ij}^{ℓ}, to a verifiable one. The general idea is that, after the protocol execution ends, each party (the Prover P)

• **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 $i \in [n]$, $\mathcal{F}_{vmpc}$ randomly generates $\vec{r}_i$. For each $i \in \mathcal{C}$, it sends $(\mathsf{randomness}, i, \vec{r}_i)$ to $\mathcal{A}_S$. At this point, $\mathcal{A}_S$ **may stop** the functionality. If it continues, then for each $i \in \mathcal{H}$ [resp $i \in \mathcal{C}$], $\mathcal{F}_{vmpc}$ gets $(\mathsf{input}, \vec{x}_i)$ from $\mathcal{Z}$ [resp. $\mathcal{A}_S$].
• **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 $\vec{m}_{ij}^{\ell}$. For all $j \in \mathcal{C}$, it sends $\vec{m}_{ij}^{\ell}$ to $\mathcal{A}_S$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\vec{m}_{ji}^{\ell}$ from $\mathcal{A}_S$.
• **After** r **rounds**, $\mathcal{F}_{vmpc}$ sends $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ 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 $\vec{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$.
$\mathcal{F}_{vmpc}$ defines $\mathcal{M} = \mathcal{B}_0 \cup \{i \in \mathcal{C} \mid \exists j \in [n], \ell \in [r'] : \vec{m}_{ij}^{\ell} \neq C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})\}$.
• **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 $\mathcal{F}_{vmpc}$ for verifiable computations

has to to prove that it followed the protocol to the set of other $n-1$ parties (the Verifiers $V_1, \ldots, V_{n-1}$). All n interactive proofs of the n provers may take place in parallel. In the rest of Sec. 4, we describe one such proof.

We assume that the majority of parties is honest. This allows us to use threshold secret sharing to make P and $V_1, \ldots, V_{n-1}$ (some of which may be corrupted) together collaborate as an honest verifier.

In the **preprocessing phase**, the parties generate *verified multiplication triples*. These are triples (a, b, c) from some ring, secret-shared among the verifiers, such that $ab = c$ and the verifiers have been convinced that this equality holds. The triples are generated and secret-shared by the prover. The verifiers execute a protocol to check that $ab = c$. Similarly, the parties generate *trusted bits*: values b from some ring, such that $b \in \{0, 1\}$ (the prover generates and shares b, and the verifiers check $b \in \{0, 1\}$). If some party misbehaves, then the preprocessing phase fails with very high probability. It is possible that this party cannot be identified.

At the beginning of the **execution phase**, P commits to its inputs and randomness by secret-sharing them among verifiers. During that phase, the parties run the protocol as usual, but they sign the messages they send out, so that the receiver may later prove which message it has got from the sender. This adds some overhead to the protocol run, but it is negligible — not many signatures are needed [47], and these can also be largely precomputed [56].

The preprocessed multiplication triples and trusted bits are not used in the execution phase. They will be needed to check the behaviour of the prover later, after the execution ends. Still, execution may not start before a sufficient number of verified triples and bits have been generated, since otherwise it will be impossible to check later if the party has cheated. Also, a party may not continue with the execution of the protocol if a signature it has received does not pass verification.

At the beginning of the **post-execution phase**, the prover commits to the messages it has sent and received during the execution phase by secret sharing them among the verifiers. The signatures generated during the execution phase do not allow the sender to deny the transmitted message without the receiver's agreement. The verifiers then repeat the computations of the prover in secret-shared manner. For additions and multiplications with constants, they use the homomorphic properties of the secret-sharing scheme. For any other operation, they use verified triples or bits to linearize it. This linearization needs the opening of some secret-shared values. The prover knows all these values and can broadcast all of them in a single round. Hence, after the commitment transformation and broadcast by prover, each verifier can compute the shares of prover's messages without further interaction.

The verification ends with the verifiers executing a protocol to check that the secret-shared messages of the prover they just computed are equal to the messages that the prover committed. At the same time, they also verify that the prover broadcast correct values. The prover sees all messages in this protocol and can complain if any verifier misbehaves. Assuming that the prover has signed all the shares that it has issued to the verifiers, the complaint can be justified. The honest majority assumption ensures that a corrupted prover cannot collaborate with the corrupted verifiers to cheat.

In the rest of this section, we describe the details of the real protocol.

4.2 Ensuring Message Delivery

Throughout the protocol execution, we meet the problem of stopping. A corrupted sender may provide an invalid signature, or even decide not to send the message at all, so that the remaining parties cannot proceed with the execution. Even if the receiver complains that it has not received the message, the remaining parties do not know whether they should blame the sender or the receiver. It would be especially sad to allow a corrupted party stop the verification phase in this way, so that the misbehaved parties will not be pinpointed.

To solve this problem, we use the transmission functionality proposed in [22]. If the receiver claims that the sender has not sent the message, then the sender has to broadcast the message, or otherwise it will be publicly blamed. This broadcast will not be used in the optimistic setting. In a single adversary model (like UC), such a broadcast does not leak any data, since if there is a conflict, then either the sender or the receiver is corrupted, and hence the adversary knows the broadcast value anyway.

We use this solution not only in the execution, but also in the preprocessing and the verification phases, in order to ensure that all the shares are delivered and all the proofs terminate.

4.3 Sharing Based Commitments

Our verification is based on a linearly homomorphic (n,t)-threshold sharing scheme that ensures consistency of the shared value and allows to prove later what has been shared. We emphasize that the initial protocol that is being verified *does not have* to be based on some linear sharing. Our verification is very generic and can verify any multiparty computation, which does not necessarily use any sharing at all. Linear sharing is needed only for the verification.

Shamir's sharing is an example of (n,t)-threshold sharing that works over any finite field. We could verify ring operations also in a finite field, but the solution would be cumbersome. A (n,t)-threshold sharing can be constructed on the basis of additive sharing. Let $a \in R$ for some ring R. Let $\mathcal{V}_1, \ldots, \mathcal{V}_{\binom{n}{t}}$ be all subsets of $[n]$ of size t. The share of each party P_k is a vector $\vec{a}^k \in R^{\binom{n}{t}}$, such that for each $j \in [\binom{n}{t}]$, the equation $\sum_{k \in \mathcal{V}_j} a_j^k = a$ holds. Also, $a_j^k = 0$ whenever $k \notin \mathcal{V}_j$.

In other words, the same value a is additively shared in $\binom{n}{t}$ different ways, each time issuing some shares $a_1, \ldots, a_t$ such that $a_1 + \ldots + a_t = a$ to a certain subset of t parties. All these $\binom{n}{t}$ sharings are independent. In this way, any t parties are able to reconstruct the secret, but less than t are not. We write $[\![a]\!] = (\vec{a}^k)_{k \in [n]}$ to denote the sharing of a.

Under honest majority assumption, a (n,t)-threshold secret sharing scheme with $t = n/2 + 1$ can be used as a commitment [19]. The committed value is shared among the n parties, and each share is signed (we assume the availability of a PKI). The commitment is opened by broadcasting the shares and verifying their signatures. If the number of honest parties is also at least t then there is an index j, such that $\mathcal{V}_j$ lists only honest parties. In this case, a set of shares can be reconstructed to at most one value, even if corrupted parties tamper with their shares (tampering may only lead to inconsistency of shares, and opening the commitment fails). The signatures on shares prevent corrupted parties from causing an inconsistent opening. Availability of at least t honest parties allows to open the commitment even if all the corrupted parties refuse to participate.

Throughout this paper, by *commitment* we mean sharing the value among the n parties using a linear (n,t)-threshold sharing. In order to avoid ambiguity, no other definition of commitment is used.

4.4 Precomputed tuples

In the preprocessing phase, the parties have to produce a sufficient number of multiplication triples and trusted bits over each ring that is used in the main protocol. The generation of such tuples is easy: the prover, allowed to know the sharings, simply generates the values itself, and commits to them by (n,t)-threshold sharing. The prover is interested in generating the tuples randomly, because his (and only his) privacy depends on it. The parties will then check whether the prover generated the tuples correctly. If the check fails, parties will not run the execution phase.

To perform the check, the parties first agree on a joint random seed. They will then perform two sub-checks: cut-and-choose and pairwise verification. In cut-and-choose, the parties randomly select k tuples and open them. This phase fails if any of the opened tuples have wrong values. Afterwards, the

parties randomly partition the remaining tuples into groups of m. In each group, they use each of the first $(m-1)$ tuples to verify the m-th one. This check fails if any of the pairwise checks fail. As analyzed below, it fails unless all tuples in a group are valid or all are invalid. After the check, the first $(m-1)$ tuples in each group are discarded and only the last one is used.

Hence, to finish the preprocessing with u tuples of certain kind, $(m \cdot u + k)$ tuples have to be generated and shared in the beginning. A combinatorial analysis (omitted due to space constraints) shows that values m and k do not need to be large. For example, if $u = 2^{20}$, then it is sufficient to take $m = 5$ and $k = 1300$. If $u = 2^{30}$ then $m = 4$ and $k = 14500$ are sufficient. These choices guarantee that if the prover aims to have an invalid tuple among the final u ones, then no strategy of generating the initial tuples makes the probability of the check succeeding greater than 2^{-80}. At the other extreme, if $u = 10$, then $m = 26$ and $k = 168$ are sufficient for the same security level.

The pairwise verification, applied to two multiplication triples, or to two trusted bits, works as follows. Note that it is certain to fail if exactly one of the tuples is a correct one.

Multiplication triples. These are triples of shared values $([\![a]\!], [\![b]\!], [\![c]\!])$ in a ring $\mathbb{Z}_n$, where a and b are random and $c = a \cdot b$. Let the triple $([\![a']\!], [\![b']\!], [\![c']\!])$ be used to verify the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. The parties compute $[\![\hat{a}]\!] = [\![a]\!] - [\![a']\!]$ and $[\![\hat{b}]\!] = [\![b]\!] - [\![b']\!]$, and declassify $\hat{a}, \hat{b}$ (the check fails if reconstruction is impossible). They will then compute $[\![z]\!] = \hat{a} \cdot [\![b]\!] + \hat{b} \cdot [\![a']\!] + [\![c']\!] - [\![c]\!]$, declassify it and check that it is 0. The check succeeds if both tuples are correct and declassifications do not fail. If one of the tuples is correct but in the other one, the third component differs from the product of the first two components by δc, then $z = \pm \delta c \neq 0$.

Trusted bits. These are shared values $[\![b]\!]$ in a ring $\mathbb{Z}_n$, where $b \in \{0, 1\}$ is random. Let the bit $[\![b']\!]$ in a ring $\mathbb{Z}_n$ be used to verify that $[\![b]\!]$ is a bit. The prover broadcasts a bit indicating whether $b = b'$ or not. If $b = b'$ was indicated, the verifiers compute $[\![z]\!] = [\![b]\!] - [\![b']\!]$, declassify it and check that it is 0. If $b \neq b'$ was indicated, the verifiers compute $[\![z]\!] = [\![b]\!] + [\![b']\!]$, declassify it and check that it is 1. In both cases, if exactly one of $[\![b]\!], [\![b']\!]$ contained a bit in $\{0, 1\}$ and the other one did not, then the check cannot succeed.

In a finite field, more efficient methods than pairwise verification are available. We discuss it in Sec. 5.3.

4.5 Commitments to messages

Before the execution phase starts, the prover P commits to its input x sharing it by a (n, t)-threshold sharing scheme.

During the execution phase, the prover P signs the outgoing messages; each message m to some P' is signed together with the identity of the protocol run it is participating in, as well as its position in this run. In protocols spanning many rounds, many signatures are necessary. To reduce the effort, methods for signing digital streams [34] may be useful.

At the start of the post-execution phase, the prover P secret-shares the message m it had sent to some P' during the execution, separately signs all shares, and sends them all to P'. Party P' (one of the verifiers) makes sure that the sharing was done correctly, and sends to each party its share, in turn signing it, so that it can be seen that both P and P' agree on the same m. Otherwise, P' publishes the message, its shares and all P's signatures, demonstrating that P has misbehaved. In this way, the sender P cannot change its mind about m, since P' is able to provide a disproof. At the same time, P' is bound to m it receives from P, since it cannot forward falsified shares without having a signature of P.

At this point, both P and P' are committed to the shares of $[\![m]\!]$ that have been issued to the honest parties. The same sharing of m is also used by P' in the proof of his correct behaviour. The sharing $[\![m]\!]$ may not correspond to m that was transmitted in the execution phase only if P and P' both are corrupted. In this case, the actual value of m during the execution phase is meaningless anyway, as it can be viewed as an inner value of the joint circuit of P and P', and it is only important that P and P' are committed to the same value m', possibly $m' \neq m$.

4.6 Commitment to randomness

Before the execution phase starts and inputs are given to the parties, the prover P must fairly choose the randomness it is going to use during the protocol, and commit to it. For this purpose, the *verifiers*

jointly generate this randomness. Each verifier V_j sends a sufficiently long random vector $\vec{r}_j$ to the prover P and commits both herself and the prover to it. The commitment is the same as for messages after the execution — V_j secret-shares the vector $\vec{r}_j$, signs the shares and sends them all to P. The prover P checks for the correctness of sharing and signatures, and forwards a signed share to each verifier. The prover P uses $\sum_j \vec{r}_j$ as its randomness. The sharing of this sum can be computed as the sum of the shares of all $\vec{r}_j$.

4.7 Verification of basic operations

A circuit (defined in Sec. 3) is composed of addition, multiplication, bit decomposition (bits), and ring transition gates (zext and trunc). We now describe how each of these gates is verified.

We have the following setup. There is an operation *op* that takes k inputs in $\mathbb{Z}_m$ and produces l outputs in $\mathbb{Z}_{m'}$. The prover knows values $x_1, \ldots, x_k$, these have been shared as $[\![x_1]\!], \ldots, [\![x_k]\!]$ among the n parties (the prover and $(n-1)$ verifiers). Moreover, the prover knows the shares of all parties. During the execution of the protocol the prover was expected to apply *op* to $x_1, \ldots, x_k$ and obtain the outputs $y_1, \ldots, y_l$. The verifiers are sure that the shares they have indeed correspond to $x_1, \ldots, x_k$ (subject to some deferred checks). A verification step gives us $[\![y_1]\!], \ldots, [\![y_l]\!]$, where the prover again knows the shares of all verifiers, but no coalition of up to $(t-1)$ verifiers has learned anything new. It also gives us a number of *alleged zeroes* — shared values $[\![z_1]\!], \ldots, [\![z_s]\!]$ (all known to prover). If $z_1 = \cdots = z_s = 0$ then the verifiers are sure that their shares of $y_1, \ldots, y_l$ indeed correspond to these values. All these equality checks are deferred to be verified (possibly succinctly) in one round later.

Verifying linear combinations. For these operations, the verifiers simply perform the corresponding operation with their shares. No alleged zeroes are created.

Verifying multiplications. The parties want to compute $[\![y]\!]$ from $[\![x_1]\!]$ and $[\![x_2]\!]$, such that $y = x_1 x_2$ in some ring $\mathbb{Z}_n$. They pick a precomputed multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ over $\mathbb{Z}_n$. The prover broadcasts $\hat{x}_1 = x_1 - a$ and $\hat{x}_2 = x_2 - b$. The parties compute $[\![y]\!] = \hat{x}_1 \cdot [\![x_2]\!] + \hat{x}_2 \cdot [\![a]\!] + [\![c]\!]$ using the homomorphic properties of the sharing scheme. Similarly, they compute the alleged zeroes $[\![z_1]\!] = [\![x_1]\!] - [\![a]\!] - \hat{x}_1$ and $[\![z_2]\!] = [\![x_2]\!] - [\![b]\!] - \hat{x}_2$.

Verifying bit decomposition. The parties want to compute $[\![y_0]\!], \ldots, [\![y_{n-1}]\!]$ from $[\![x]\!]$, where $x \in \mathbb{Z}_{2^n}$, $y_i \in \{0,1\}$, $x = \sum_{i=0}^{n-1} 2^i y_i$ and all sharings are over $\mathbb{Z}_{2^n}$. They pick n trusted bits $[\![b_0]\!], \ldots, [\![b_{n-1}]\!]$, shared over $\mathbb{Z}_{2^n}$. The prover broadcasts bits $c_0, \ldots, c_{n-1}$. The parties take $[\![y_i]\!] = [\![b_i]\!]$ if $c_i = 0$, and $[\![y_i]\!] = 1 - [\![b_i]\!]$, if $c_i = 1$ (this explains how the prover computes $c_0, \ldots, c_{n-1}$). The parties compute the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=0}^{n-1} 2^i \cdot [\![y_i]\!]$.

Verifying conversions between rings. The parties want to compute $[\![y]\!]$ from $[\![x]\!]$, such that $y = x$, but while the sharing of x, is over $\mathbb{Z}_{2^n}$, the sharing of y is over $\mathbb{Z}_{2^m}$. If $m < n$, then the parties simply drop $n-m$ highest bits from all shares of x, resulting in shares of y. We denote this operation by $[\![y]\!] = \mathsf{trunc}([\![x]\!])$. Otherwise, the parties perform the bit decomposition of $[\![x]\!]$ as in previous paragraph, obtaining the shared bits $[\![y'_0]\!], \ldots, [\![y'_{n-1}]\!]$; the bits are shared over the ring $\mathbb{Z}_{2^m}$. They will then compute $[\![y]\!] = \sum_{i=0}^{n-1} 2^i \cdot [\![y'_i]\!]$ and the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=1}^{n-1} 2^i \cdot \mathsf{trunc}([\![y'_i]\!])$.

Verifying outputs of circuits. By composing the steps described above, the parties obtain a sharing $[\![y]\!]$ of some output of the circuit from the commitments to its inputs. The prover has previously committed that output as $[\![y']\!]$. To verify the correctness of prover's commitment, the parties simply produce an alleged zero $[\![z]\!] = [\![y]\!] - [\![y']\!]$.

In the verification of operations, the communication between parties (if any) only originated from the prover. Thus the verification of a circuit can be performed by the prover first broadcasting a single long message, followed by all parties performing local computations.

4.8 Checking of alleged zeroes

An alleged zero $[\![z]\!]$ is verified by simply opening the secret sharing. After the opening, each party may reconstruct z and verify that it is equal to 0. This opening preserves prover's privacy because each $[\![z]\!]$ is just a random sharing of 0 if the prover behaved honestly.

The opening is simplified by the prover knowing all shares of $[\![z]\!]$. He sends all shares to all verifiers (signed). A verifier complains if the received shares do not combine to 0, or if its own share in $[\![z]\!]$ is different from the one received from the prover. In both cases, the verifier publishes the shares signed by the prover. In the former case, the prover's maliciousness is immediately demonstrated. In the latter

$\mathcal{F}_{verify}$ works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ (the latter is used only for message commitments), the compound operation $f(id)$ to verify (a composition of basic operations of Sec. 4.7), the input indices $\vec{xid}(id)$, and the output indices $\vec{yid}(id)$ on which $f(id)$ should be verified. The committed values are stored in an array *comm*. The messages are first stored in an array *sent* before they are finally committed. Let $\mathcal{A}_S$ denote the ideal adversary.

Initialization: On input $(\mathsf{init}, f(\cdot), \vec{xid}(\cdot), \vec{yid}(\cdot), p(\cdot), p'(\cdot))$ from all the (honest) parties, initialize *comm* and *sent* to empty arrays. Assign the mappings $f := f(\cdot)$, $\vec{xid} := \vec{xid}(\cdot)$, $\vec{yid} := \vec{yid}(\cdot)$, $p := p(\cdot)$, $p' := p'(\cdot)$.

Input Commitment: On input $(\mathsf{commit\_input}, \vec{x}, id)$ from $P_{p(id)}$, and $(\mathsf{commit\_input}, id)$ from all honest parties, check if $comm[id]$ exists. If it does, then do nothing. Otherwise, assign $comm[id] := \vec{x}$. If $p(id) \in \mathcal{C}$, then $\vec{x}$ is chosen by $\mathcal{A}_S$.

Message Commitment: On input $(\mathsf{send\_msg}, \vec{m}, id)$ from $P_{p(id)}$, output $\vec{m}$ to $P_{p'(id)}$. If $p(id) \in \mathcal{C}$, then $\vec{m}$ is chosen by $\mathcal{A}_S$. If $p'(id) \in \mathcal{C}$, output $\vec{m}$ to $\mathcal{A}_S$. Assign $sent[id] := \vec{m}$.

On input $(\mathsf{commit\_msg}, id)$ from all honest parties, check if $sent[id]$ and $comm[id]$ are defined. If either $comm[id]$ is defined, or $sent[id]$ is not defined, then do nothing. Otherwise, assign $comm[id] := sent[id]$. If both $p(id), p'(id) \in \mathcal{C}$, assign $comm[id] := \vec{m}^*$, where $\vec{m}^*$ is chosen by $\mathcal{A}_S$.

Randomness Commitment: On input $(\mathsf{commit\_rnd}, id)$ from $P_{p(id)}$, and $(\mathsf{commit\_rnd}, id)$ from all honest parties, check if $comm[id]$ exists. If it does, then do nothing. Otherwise, generate a random $\vec{r}$, and assign $comm[id] := \vec{r}$. Output $\vec{r}$ to $P_{p(id)}$. If $p(id) \in \mathcal{C}$, output $\vec{r}$ also to $\mathcal{A}_S$.

Verification: On input (verify, id) from all honest parties, take $\vec{x} = (comm[i])_{i \in \vec{xid}(id)}$ and $\vec{y} = (comm[i])_{i \in \vec{yid}(id)}$. For $f := f(id)$, compute $\vec{y'} = f(\vec{x})$. Output the difference $\vec{y'} - \vec{y}$, to each party and $\mathcal{A}_S$.

Stopping: On input (stop, k) from $\mathcal{A}_S$ for $k \in \mathcal{C}$, stop the functionality and output $(\mathsf{corrupt}, k)$ to each party.

Figure 2: Ideal functionality $\mathcal{F}_{verify}$

case, note that an alleged zero $[\![z]\!]$ is a linear combination (with public coefficients) of secret-shared values, where all shares are signed by the prover. The complaining verifier also publishes its shares of all values from which it computed its share in $[\![z]\!]$. All other verifiers can now repeat the computation and check whether it was correctly performed.

Similarly to Sec. 4.7, all communication in the checking step also originates from the prover, unless there are complaints. Hence the messages in these two steps can be transmitted in the same round, and the whole post-execution phase, in the case of no complaints, only requires two rounds of communication. The transformation of commitments takes place in both rounds, while the messages required for verifying basic operations and checking alleged zeroes are broadcast during the second round.

4.9 Putting it all together

From the informal constructions of Sec. 4.2- 4.8, we may abstract away the functionality $\mathcal{F}_{verify}$ that we will use to verify circuit operations with respect to the committed inputs, randomness, and communication. It is given in Fig. 2. The implementation of $\mathcal{F}_{verify}$ is built on top of precomputed multiplication triples and trusted bits. Their generation is abstracted away into a functionality $\mathcal{F}_{pre}$ given in Fig. 3.

Lemma 1 *Let $\mathcal{C}$ be the set of corrupted parties. Assuming the existence of PKI and $|\mathcal{C}| < n/2$, there exists a protocol Π_{pre} UC-realizing $\mathcal{F}_{pre}$.*

Proof sketch: The construction of Π_{pre} follows Sec. 4.4. It is easy to see that the protocol does not leak any information about the triples and bits that are finally chosen, since they are masked with unifomly distributed values that are never published. The cut-and-choose and pairwise check ensure that the remaining triples and bits are correct with overwhelming probability. This protocol works similary to [25], and we refer to [25] for a more formal proof. □

Lemma 2 *Let $\mathcal{C}$ be the set of corrupted parties. Assuming the existence of PKI and $|\mathcal{C}| < n/2$, there exists a protocol Π_{verify} UC-realizing $\mathcal{F}_{verify}$ in $\mathcal{F}_{pre}$-hybrid model.*

$\mathcal{F}_{pre}$ works with unique identifiers id, encoding a ring size $m(id)$ in which the tuples are shared, the party $p(id)$ that gets all the shares, and the number $n(id)$ of tuples to be generated. It stores a vector *triple* of precomputed triples, and a vector *bit* of trusted bits. Let $\mathcal{A}_S$ denote the ideal adversary.
Initialization: On input $(\mathsf{init}, m(\cdot), n(\cdot), p(\cdot))$ from each (honest) party, initialize *triple* and *bit* to empty arrays. Assign the functions $m := m(\cdot)$, $n := n(\cdot)$, $p := p(\cdot)$.
Multiplication triple generation: On input (triple, id) from P_i, check if $triple[id]$ exists. If it does, take $(\vec{r}_x^k, \vec{r}_y^k, \vec{r}_{xy}^k)_{k\in[n]} := triple[id]$. Otherwise, generate $\vec{r}_x \xleftarrow{\$} \mathbb{Z}_{m(id)}^{n(id)}$, $\vec{r}_y \xleftarrow{\$} \mathbb{Z}_{m(id)}^{n(id)}$, and compute $\vec{r}_{xy} = \vec{r}_x \cdot \vec{r}_y$. Compute the shares $(\vec{r}_x^k)_{k\in[n]}$, $(\vec{r}_y^k)_{k\in[n]}$, and $(\vec{r}_{xy}^k)_{k\in[n]}$ of $\vec{r}_x$, $\vec{r}_y$, and $\vec{r}_{xy}$ respectively. Assign $triple[id] := (\vec{r}_x^k, \vec{r}_y^k, \vec{r}_{xy}^k)_{k\in[n]}$. If $p(id) \neq i$, send $(\vec{r}_x^i, \vec{r}_y^i, \vec{r}_{xy}^i)$ to P_i. Otherwise, send $(\vec{r}_x^k, \vec{r}_y^k, \vec{r}_{xy}^k)_{k\in[n]}$ to P_i. For all $k \in \mathcal{C}$, send $(\vec{r}_x^k, \vec{r}_y^k, \vec{r}_{xy}^k)$ also to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send all shares $(\vec{r}_x^k, \vec{r}_y^k, \vec{r}_{xy}^k)_{k\in[n]}$ to $\mathcal{A}_S$.
Trusted bit generation: On input (bit, id) from P_i, check if $bit[id]$ exists. If it does, take $(\vec{r}^k)_{k\in[n]} := bit[id]$. Otherwise, generate a vector of random bits $\vec{r} \xleftarrow{\$} \mathbb{Z}_2^{n(id)}$. Compute the shares $(\vec{r}^k)_{k\in[n]}$ of $\vec{r}$ over $\mathbb{Z}_{m(id)}^{n(id)}$. Assign $bit[id] := (\vec{r}^k)_{k\in[n]}$. Handle $(\vec{r}^k)_{k\in[n]}$ similarly to the multiplication triple shares.
Stopping: On input (stop) from $\mathcal{A}_S$, stop the functionality and output $\perp$ to all parties.

Figure 3: Ideal functionality $\mathcal{F}_{pre}$

Proof sketch: We describe the protocol Π_{verify} and the simulator S that translates between the messages $\mathcal{F}_{verify}$ exchanges with the ideal adversary $\mathcal{A}_S$, and the messages the protocol Π_{verify} exchanges with the real adversary $\mathcal{A}$ over the network. The task of S is to make $\mathcal{Z}$ believe that it is executing Π_{verify}, while it is actually executing $\mathcal{F}_{verify}$.

All the commitments of $\mathcal{F}_{verify}$ can be realized by (n,t)-threshold sharing, which works on the assumption $|\mathcal{C}| < n/2$ (see Sec. 4.3). As discussed in Sec. 4.2, assuming $|\mathcal{C}| < n/2$ we can ensure that, either all shares are delivered and provided with valid signatures, or the cheating party P_k will be publicly blamed. In the latter case, the simulator S sends (stop, k) to $\mathcal{F}_{verify}$, causing it to output $(\mathsf{corrupt}, k)$ to each party. Since $\mathcal{A}$ obtains at most $t-1$ shares issued to the corrupted parties, they are distributed uniformly, and so S may sample the shares of honest parties randomly.

The shares of corrupted parties are chosen by $\mathcal{A}$. If the shares are inconsistent, there is no way to check it in Π_{verify} immediately. At the same time, $\mathcal{F}_{verify}$ is waiting for a commitment from $\mathcal{A}_S$. Here S may reconstruct the commitment from any set of shares belonging to some t honest parties. As we show later, the simulation works with an arbitrary set of t honest parties, even if the sharing is inconsistent. For simplicity, in the remaining proof, instead of writing that S has received shares of x from $\mathcal{A}$, we write that S has received x from $\mathcal{A}$, assuming that x has been reconstructed from shares of a certain set of t honest parties.

Initialization: Before the inputs are given to the parties, a certain number of multiplication triples and bits should be generated. Each (honest) party sends $(\mathsf{init}, m(\cdot), n(\cdot), p(\cdot))$ to $\mathcal{F}_{pre}$, where $m(\cdot)$,$n(\cdot)$,$p(\cdot)$ are determined by the function descriptions $f(\cdot)$, $\vec{xid}(\cdot)$,$\vec{yid}(\cdot)$,$p(\cdot)$,$p'(\cdot)$ received by the parties from the environment, that fully describe the particular operations that are going to be verified, and the responsible prover parties. The mappings $m(\cdot)$,$n(\cdot)$,$p(\cdot)$ are all defined over the same domain, which is a set of identifiers id. For each such id, each (honest) party sends to $\mathcal{F}_{pre}$ a call (triple, id) or (bit, id) (the choice of triple and bit is defined by $f(\cdot)$).

Inputs: The inputs of $P_{p(id)}$ are committed by sharing, without doing any additional checks. The inputs of corrupted parties are chosen by $\mathcal{A}$, and S forwards them to $\mathcal{F}_{verify}$.

Messages: First of all, the sender $P = P_{p(id)}$ signs m and sends it to the receiver $P' = P_{p'(id)}$. The delivery of signed m is ensured by the message transmission functionality (Sec. 4.2). This intermediate step is only needed to allow to postpone the sharing, so that the execution phase does not become more expensive. At this point, if P is corrupted, then S delivers to $\mathcal{F}_{verify}$ the value m that is chosen by $\mathcal{A}$.

Later, P shares m, signs the shares, and sends them to P' who finally distributes them among the other parties (Sec. 4.5). Requiring signatures on shares from both the sender and the receiver ensures that P and P' agree on the same committed message m. Since P' holds the signature of P on m, even if P does not share m correctly, P' may commit m itself by publishing it, so that the parties may share the published m themselves in an arbitrary pre-agreed way. Publishing m can be easily simulated by S, since if at least one of P and P' is corrupted, then it has already seen m that was either chosen by $\mathcal{A}$, or was sent to S by $\mathcal{F}_{verify}$.

Randomness: For the randomness commitment, each verifier V_j provides a uniformly distributed vector $\vec{r}_j$ that will be used in the sum $\vec{r} = \vec{r}_1 + \cdots + \vec{r}_{n-1}$ (Sec. 4.6). Since at least one verifier is honest, the sum is uniformly distributed even if the corrupted parties try to tamper with their contributions. After S receives from $\mathcal{A}$ all the vectors $\vec{r}_j$ for $j \in \mathcal{C}$, it generates the remaining shares of at least one honest party in such a way that $\vec{r}_1 + \cdots + \vec{r}_{n-1}$ equals to $\vec{r}$ chosen by $\mathcal{F}_{verify}$.
Verification: The preprocessed multiplication triples and bits are used to linearize all basic operations and their compositions (Sec. 4.7). For this, the prover P first needs to publish the values $\hat{x}_1 = x_1 - a$ and $\hat{x}_2 = x_2 - b$ for each multiplication triple (a, b, c), and $c_i \in \{b_i, 1 - b_i\}$ for each trusted bit b_i. For corrupted provers, these values are chosen by $\mathcal{A}$. For honest provers, the values a, b, b_i have not been used in the simulations yet, and they are still uniformly distributed. In this way, they work as masks, and the values $\hat{x}_1$, $\hat{x}_2$, c_i may be sampled by S from a uniform distribution (where $c_i \in \{0, 1\}$ is a random bit).

For any composition of basic operations f, each verifier V_k may now locally compute $\vec{z}'^k = f(\vec{x}'^k) - \vec{y}'^k$ on its shares, producing a vector of alleged zeroes $[\![\vec{z}]\!]$. The parties need to check if $\vec{z} = \vec{0}$. For this, the prover P first computes all the shares $\vec{z}^{*k}$ itself, signs, and sends them to each other party (Sec. 4.8). The definition of $\mathcal{F}_{verify}$ allows to reveal $\vec{z}$ to $\mathcal{A}_S$, so $\vec{z}$ (and its shares) of honest provers can also be simulated by S. If any V_k sees that $\vec{z}^{*k} \neq \vec{z}'^k$, then it publishes all the shares $\vec{x}'^k$ and $\vec{y}'^k$, so that each other party may now compute $\vec{z}'^k = f(\vec{x}'^k) - \vec{y}'^k$ itself. Revealing $\vec{x}'^k$ and $\vec{y}'^k$ can be easily simulated, since in the case of a conflict they are known by $\mathcal{A}$ (and hence S) anyway. Since all $\vec{x}'^k$ and $\vec{y}'^k$ have been signed, a corrupted verifier cannot blame an honest prover. If V_k and P are both corrupted, they are able to present $\vec{z}^{*k} \neq \vec{z}'^k$ to the other verifiers who cannot check the validity of $\vec{z}^{*k}$. However, since at least t parties are honest, and we are using (n, t)-threshold sharing, there will be at least t shares $\vec{z}^{*k} = \vec{z}$ that uniquely determine the value $\vec{z}^* = \vec{z}$ that is reconstructed from these t shares. Introducing any tampered shares $\vec{z}^{*k} \neq \vec{z}'^k$ will make the sharing inconsistent, and the proof will not be accepted. In this case, S sends $(\mathsf{stop}, p(id))$ to $\mathcal{F}_{verify}$. Hence the verification is indeed bound to $\vec{x}$ and $\vec{y}$ committed before, and if the shares are consistent, they correspond to the same $\vec{x}$ and $\vec{y}$ that S reconstructed from the shares of some t honest parties and committed to $\mathcal{F}_{verify}$ before.
Stopping: $\mathcal{F}_{verify}$ allows $\mathcal{A}_S$ to blame any party P_k for $k \in \mathcal{C}$. In Π_{verify}, the accusations may take place only if some party presents a signature, proving that some other party has generated a set of shares that does not correspond to the previously signed message (during commitments), or it has generated a share that does not correspond to the shares it has committed before (during the verification). In both cases, an honest party would never sign a value that does not correspond to its previous signatures. Assuming a good PKI, accusing P_k for $k \notin \mathcal{C}$ may happen only with a negligible probability. □

Having an implementation of $\mathcal{F}_{verify}$, we may now state and prove the main theorem of this paper.

Theorem 1 *Let $\mathcal{C}$ be the set of corrupted parties. There exists a protocol Π_{vmpc} UC-emulating $\mathcal{F}_{vmpc}$ in $\mathcal{F}_{verify}$-hybrid model if $|\mathcal{C}| < n/2$.*

Proof sketch: The real protocol Π_{vmpc} follows the outline of Sec. 4.1. The parties $P_1, \ldots, P_n$ work as follows.

In the preprocessing phase, each (honest) party sends $(\mathsf{init}, f(\cdot), \vec{xid}(\cdot), \vec{yid}(\cdot), p(\cdot), p'(\cdot))$ to $\mathcal{F}_{verify}$, where the argument functions are defined by the circuits that the parties get from the environment. After that, each (honest) party sends $(\mathsf{commit\_rnd}, id)$ to $\mathcal{F}_{verify}$ to generate the committed randomness for $P_{p(id)}$. By definition of $\mathcal{F}_{verify}$, the randomness comes from a uniform distribution, as the randomness generated by $\mathcal{F}_{vmpc}$.

After the inputs are given to the parties by $\mathcal{Z}$, $P_{p(id)}$ commits to its inputs by sending $(\mathsf{commit\_input}, \vec{x}, id)$ to $\mathcal{F}_{verify}$, and each (honest) P_i supports it by sending $(\mathsf{commit\_input}, id)$ to $\mathcal{F}_{verify}$. The corrupted parties may commit arbitrary values, which is allowed by definition of $\mathcal{F}_{vmpc}$. The simulator S obtains the inputs of corrupted parties from $\mathcal{A}$ and forwards them to $\mathcal{F}_{vmpc}$.

During the execution phase, for each message identifier id, the (honest) sender $P_{p(id)}$ sends $(\mathsf{send\_msg}, \vec{m}, id)$ to $\mathcal{F}_{verify}$. If at least the sender $P = P_{p(id)}$ or the receiver $P' = P_{p'(id)}$ is honest, either the protocol stops and P gets publicly blamed, or $\mathcal{F}_{verify}$ delivers $\vec{m}$ to P'. If P is honest, then it delivers the same $\vec{m}$ that is computed by $\mathcal{F}_{vmpc}$. If P is corrupted, but P' is honest, then $\vec{m}$ can be arbitrary, and for $\mathcal{F}_{vmpc}$ it is chosen by $\mathcal{A}_S$, so S delivers to $\mathcal{F}_{vmpc}$ the message $\vec{m}$ chosen by $\mathcal{A}$. If both P and P' are corrupted, then they may agree on committing $\vec{m}^* \neq \vec{m}$ later, so it cannot be fixed yet, but in this case $\mathcal{F}_{vmpc}$ also does not expect $\vec{m}$ yet.

After the execution phase, each (honest) party sends $(\mathsf{commit\_msg}, id)$ to $\mathcal{F}_{verify}$, committing the sender $P_{p(id)}$ and the receiver $P_{p'(id)}$ to the exchanged message $\vec{m}$ that $\mathcal{F}_{verify}$ has received at some point earlier. If both $P_{p(id)}$ and $P_{p'(id)}$ are corrupted, they may commit to an arbitrary $\vec{m}^*$. $\mathcal{F}_{vmpc}$ also expects all such messages $\vec{m}^*$ from $\mathcal{A}_S$ at this point, so S just delivers $\vec{m}^*$ from $\mathcal{A}$ to $\mathcal{F}_{vmpc}$.

In the verification phase, each output of the prover's circuit (including the outgoing messages) is viewed as a function f applied to the inputs, randomness, and the incoming messages committed so far (as in the definition of $\mathcal{F}_{vmpc}$). In Sec. 4.7, we have provided verifications for the linear combinations and multiplications (which are sufficient to compute any arithmetic circuit over a ring), and also the bit decomposition and the ring conversion (which are sufficient to compute any combined computation over several rings). We also have shown in Sec. 4.7 that the outputs of these operations are generated locally by the verifiers on-the-fly. Hence f can be represented as a composition of the basic operations of Sec. 4.7, and we may use $\mathcal{F}_{verify}$ to verify the computation of f on the quantities that have already been committed. By definition, $\mathcal{F}_{verify}$ reveals the difference $f(\vec{x}) - \vec{y}$, where $\vec{x}$ and $\vec{y}$ are the values committed so far. If the prover is honest, then he has behaved correctly in the execution phase, and hence $f(\vec{x}) - \vec{y} = \vec{0}$. Publishing $\vec{0}$ does not reveal any private information. If the prover is corrupted, then the difference $f(\vec{x}) - \vec{y}$ is known by the adversary anyway. Both cases can be simulated by S straightforwardly.

If $\mathcal{F}_{verify}$ outputs 0 for each output of all the prover P_k's circuits, then the proof of P_k is accepted. Otherwise, each (honest) party claims that P_k has cheated, and it will be accused by Byzantine agreement. The set of cheated parties is captured by the set $\mathcal{M}$ constructed by $\mathcal{F}_{vmpc}$. In addition, $\mathcal{F}_{vmpc}$ allows $\mathcal{A}_S$ to include more corrupted parties of $\mathcal{C}$ into the set of blamed parties $\mathcal{B}_i$ that it finally outputs to P_i. These additional parties are those that attempted to misbehave while using $\mathcal{F}_{verify}$, causing messages $(\mathsf{corrupt}, k)$ to be output. By definition of $\mathcal{F}_{verify}$, such messages can be output only for $k \in \mathcal{C}$. Hence we have $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$, as for $\mathcal{F}_{vmpc}$. □

5 Extensions

In this section, we describe possible optimizations and extensions of the protocol described in Sec. 4.

5.1 Optimizations

Our protocols allow a general optimization: in a secret sharing $[\![a]\!] = (\vec{a}^k)_{k\in[n]}$ we can delete from the vectors $\vec{a}^k$ (of length $\binom{n}{t}$) the components that correspond to the subsets of $[n]$ the contain (the index of) the prover, because the prover knows all the shares anyway and can make up its own only when required to send it to someone. Effectively, this means that $[\![a]\!]$ is shared among the $n-1$ verifiers using $(n-1,t)$-threshold sharing scheme described in Sec. 4.3. In particular, if $n = 3$ (as in Sharemind), then $[\![a]\!] = (a_1, a_2)$, where $a_1 + a_2 = a$ in some ring R and a_i is held by the i-th verifier. This simplification enables many more optimizations for $n = 3$, as described below.

Sharing the messages. At the beginning of the post-execution phase, to share the messages it had sent or received during the execution phase, the prover does not have to do anything: the messages are already shared. Indeed, one of the verifiers, being the recipient or the sender of that message, already knows it. The other verifier's share of that message will be 0.

Committing to randomness. For the prover to commit to its randomness at the beginning of the execution phase, he receives a signed random seed s_i from the i-th verifier. He will then use $G(s_1)+G(s_2)$ (for the PRG G) as its randomness.

In case of Sharemind, the commitment is even simpler. In all protocols currently in use, any random value is known by exactly two parties out of three (each pair of parties has a common random seed). Hence any random value r used by the prover is already shared in the same manner as the messages.

Checking alleged zeroes. To check if $[\![z^1]\!], \ldots, [\![z^s]\!]$ are all equal to 0, the first verifier computes $H(z_1^1, z_1^2, \cdots, z_1^s)$ and the second verifier computes $H((-z_2^1), (-z_2^2), \cdots, (-z_2^s))$, where H is a hash function and z_1^i, z_2^i are the shares of $[\![z^i]\!]$ held by the first and second verifier, respectively. They sign and send the computed shares to each other and to the prover, and check that they are equal.

5.2 Other operations

The circuits for computing the messages in certain protocols of Sharemind use some more operations in addition to those described in Sec. 4.7. We now describe their verification. Note that the multiplication protocol only needs multiplications to be verified [11, Alg. 2].

Comparison. The computation of a shared bit $[\![y]\!]$ from $[\![x_1]\!], [\![x_2]\!] \in \mathbb{Z}_{2^n}$, indicating whether $x_1 < x_2$, proceeds by the following composition. First, convert the inputs to the ring $\mathbb{Z}_{2^{n+1}}$, let the results be $[\![x_1']\!]$ and $[\![x_2']\!]$. Next, compute $[\![w]\!] = [\![x_1']\!] - [\![x_2']\!]$ in the ring $\mathbb{Z}_{2^{n+1}}$. Finally, decompose $[\![w]\!]$ into bits and let $[\![y]\!]$ be the highest bit.

Bit shifts. To compute $[\![y]\!] = [\![x]\!] \ll [\![x']\!]$, where $[\![y]\!]$ and $[\![x]\!]$ are shared over $\mathbb{Z}_{2^n}$ and $[\![x']\!]$ is shared over $\mathbb{Z}_n$, the parties need a precomputed *characteristic vector* (CV) tuple $([\![r]\!], [\![\vec{s}]\!])$, where $[\![r]\!]$ is shared over $\mathbb{Z}_n$, $[\![s_i]\!]$ are shared over $\mathbb{Z}_{2^n}$, the values s_i are bits, the length of $\vec{s}$ is n, and $s_i = 1$ iff $i = r$. The prover broadcasts $\hat{x} = r - x' \in \mathbb{Z}_n$. The verifiers compute $[\![\vec{s}']\!] = \mathsf{rot}(\hat{x}, [\![\vec{s}]\!])$, defined by $[\![s_i']\!] = [\![s_{(i+\hat{x}) \bmod n}]\!]$ for all $i < n$. Note that $s_i' = 1$ iff $i = x'$. The verifiers compute $[\![2^{x'}]\!] = \sum_{i=0}^{n-1} 2^i [\![s_i']\!]$ and multiply it with $[\![x]\!]$ (using a multiplication triple). They compute the alleged zero $[\![z]\!] = [\![r]\!] - [\![x']\!] - \hat{x}$, as well as two alleged zeroes from the multiplication.

To compute $[\![y]\!] = [\![x]\!] \gg [\![x']\!]$, the parties first reverse $[\![x]\!]$, using bit decomposition. They will shift the reversed value left by $[\![x']\!]$ positions, and reverse the result again.

During precomputation phase, the CV tuples have to be generated. Their correctness control follows Sec. 4.4, with the following pairwise verification operation. Given tuples $([\![r]\!], [\![\vec{s}]\!])$ and $([\![r']\!], [\![\vec{s}']\!])$, the verifiers compute $[\![\hat{r}]\!] = [\![r']\!] - [\![r]\!]$, declassify it, compute $[\![\hat{\vec{s}}]\!] = [\![\vec{s}]\!] - \mathsf{rot}(\hat{r}, [\![\vec{s}']\!])$, declassify it and check that it is a vector of zeroes. Recall (Sec. 4.4) that we need the pairwise verification to only point out whether one tuple is correct and the other one is not.

Rotation. The computation of $[\![\vec{y}]\!] = \mathsf{rot}([\![x']\!], [\![\vec{x}]\!])$ for $[\![\vec{x}]\!], [\![\vec{y}]\!] \in \mathbb{Z}_{2^n}^m$ and $[\![x']\!] \in \mathbb{Z}_m$ could be built from bit shifts, but a direct computation is more efficient. The parties need a *rotation tuple* $([\![r]\!], [\![\vec{s}]\!], [\![\vec{a}]\!], [\![\vec{b}]\!])$, where $[\![r]\!]$ and $[\![\vec{s}]\!]$ are a CV tuple (with $r \in \mathbb{Z}_m$ and $\vec{s} \in \mathbb{Z}_{2^n}^m$), $\vec{a} \in \mathbb{Z}_{2^n}^m$ is random and the elements of $\vec{b}$ satisfy $b_i = a_{(i+r) \bmod m}$.

The prover broadcasts $\hat{r} = x' - r$ and $\hat{\vec{x}} = \vec{x} - \vec{a}$. The verifiers can now compute

$$[\![c_i]\!] = \hat{\vec{x}} \cdot \mathsf{rot}(i, [\![\vec{s}]\!]) \qquad (i \in \{0, \ldots, m-1\})$$
$$[\![\vec{y}]\!] = \mathsf{rot}(\hat{r}, [\![\vec{c}]\!]) + \mathsf{rot}(\hat{r}, [\![\vec{b}]\!]) \ .$$

Here $\cdot$ denotes the scalar product; each c_i is equal to some $\hat{x}_i$. The correctness of the computation follows from $\vec{c} = \mathsf{rot}(r, \hat{\vec{x}})$. The procedure gives the alleged zeroes $[\![z']\!] = [\![x']\!] - [\![r]\!] - \hat{r}$ and $[\![\vec{z}]\!] = [\![\vec{x}]\!] - [\![\vec{a}]\!] - \hat{\vec{x}}$.

The pairwise verification of rotation tuples
$\mathbf{T} = ([\![r]\!], [\![\vec{s}]\!], [\![\vec{a}]\!], [\![\vec{b}]\!])$ and $\mathbf{T}' = ([\![r']\!], [\![\vec{s}']\!], [\![\vec{a}']\!], [\![\vec{b}']\!])$ works similarly, using the tuple $\mathbf{T}'$ to rotate $[\![\vec{a}]\!]$ by $[\![r]\!]$ positions and checking that the result is equal to $[\![\vec{b}]\!]$. Additionally, pairwise verification of CV tuples is performed on $([\![r]\!], [\![\vec{s}]\!])$ and $([\![r']\!], [\![\vec{s}']\!])$.

Shuffle. The parties want to apply a permutation σ to a vector $[\![\vec{x}]\!] \in \mathbb{Z}_n^m$, obtaining $[\![\vec{y}]\!]$ satisfying $y_i = x_{\sigma(i)}$. Here $\sigma \in S_m$ is known to the prover and to exactly one of the verifiers [45]. To protect prover's privacy, it must not become known to the other verifier. In the following, we write $[\sigma]$ to denote that σ is known to the prover and to one of the verifiers (w.l.o.g., to verifier V_1).

The parties need a precomputed *permutation triple*
$([\rho], [\![\vec{a}]\!], [\![\vec{b}]\!])$, where $\rho \in S_m$, $\vec{a}, \vec{b} \in \mathbb{Z}_n^m$ and $\vec{b} = \rho(\vec{a})$. Both the prover and verifier V_1 sign and send $\tau = \sigma \circ \rho^{-1}$ to V_2 (one of them may send $H(\tau)$; verifier V_2 complains if received τ-s are different). The prover broadcasts $\hat{\vec{x}} = \vec{x} - \vec{a}$. The verifiers compute their shares $(\vec{y}_1, \vec{y}_2)$ of $[\![\vec{y}]\!]$ as $\vec{y}_1 = \tau(\vec{b}_1 + \rho(\hat{\vec{x}}))$ and $\vec{y}_2 = \tau(\vec{b}_2)$, where $\vec{b}_i$ is the i-th verifier's share of $[\![\vec{b}]\!]$. The alleged zeroes $[\![\vec{z}]\!] = [\![\vec{x}]\!] - [\![\vec{a}]\!] - \hat{\vec{x}}$ are produced.

5.3 Preprocessing phase for Finite Fields

Over a finite field, then we may use more efficient methods for generating the preprocessed tuples of Sec. 4.4. We can replace the cut-and-choose and pairwise verification steps with an application of linear error correcting codes [5]. This technique allows the construction of n verified triples from only $n + k$ initial ones, where k is proportional to η. Hence for large values of n, the communication cost due to verification is negligible.

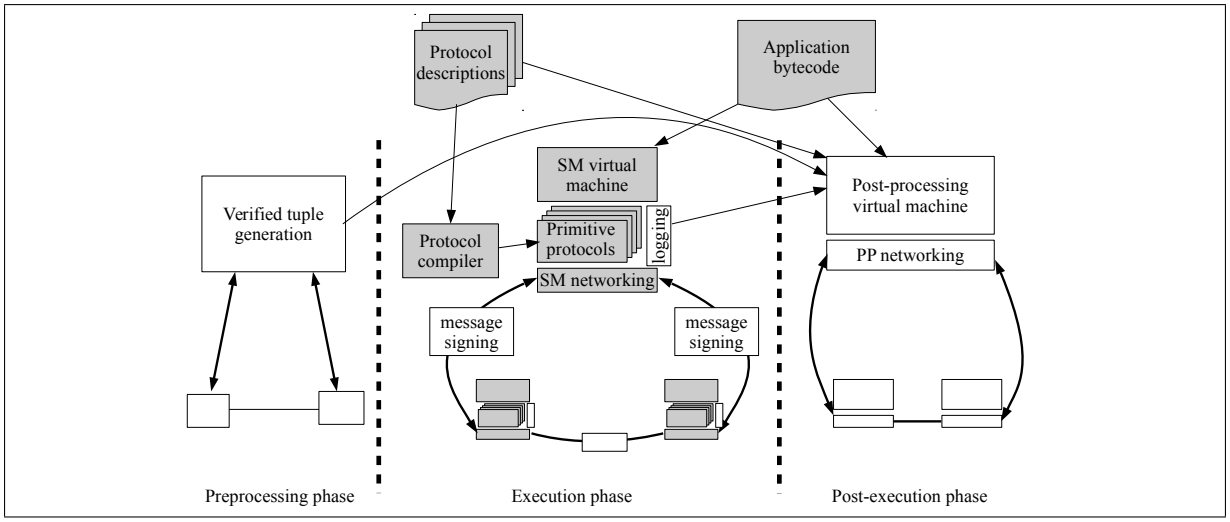

Figure 4: Components of Sharemind with verification

5.4 Auditability

If a party P has deviated from the protocol, then all honest parties will learn its identity during the post-execution phase. In this case, assuming that P does not drop out from the verification process at all, the honest parties are going to have a set of statements signed by P, pertaining to the values of various messages during the preprocessing, execution, and post-execution phases, from which the contradiction can be derived. These statements may be presented to a judge that is trusted to preserve the privacy of honest parties.

6 Evaluation

6.1 Implementation

We have implemented the verification of computations for the Sharemind protocol set [11,39,42,45]. The previously existing (gray) and newly implemented (white) components are depicted in Fig. 4.

Sharemind has a large protocol set for integer, fix- and floating point operations, as well as for shuffling the arrays, that can be used by a privacy-preserving application. Almost all these protocols are generated from a clear description, how messages are computed and exchanged between parties [44]. The application itself is described in a high-level language that is compiled into bytecode [9], instructing the Sharemind virtual machine to call the lower-level protocols in certain order with certain arguments. Both descriptions are used in the post-execution phase.

Preprocessing phase. The verified tuple generator has been implemented in C, compiled with `gcc` ver. 4.8.4, using `-O3` optimization level, and linking against the cryptographic library of OpenSSL 1.0.1k. We have tried to simplify the communication pattern of the tuple generator as much as possible, believing it to maximize performance. On the other hand, we have not tried to parallelize the generator, neither its computation, nor the interplay of computation and communication. Hence we believe that further optimizations are possible.

The generator works as follows. If the parties want to produce n verified tuples, then (i) they will select m and k appropriately for the desired security level (Sec. 4.4); (ii) the prover sends shares of $(mn + k)$ tuples to verifiers; (iii) verifiers agree on a random seed (used to determine, which tuples are opened and which are grouped together) and send it back to the prover; (iv) prover sends to the verifiers k tuples that were to be opened, as well as the differences between components of tuples that are needed for pairwise verification; (v) verifiers check the well-formedness of opened tuples and check the alleged zeroes stating that they received from the prover the same values, these values match the tuples, and the pairwise checks go through. Steps (ii) and (iv) are communication intensive. In step (iii), each verifier generates a short random vector and sends it to both the prover and the other verifier. The concatenation of these vectors is used as the random seed for step (iv). Step (v) involves the verifiers comparing that

Table 1: Time to generate $n = 10^8$ verified tuples for security parameter $\eta = 80$ ($m = 4$, $k = 15000$)

Tuple	**width**	**time**
Multiplication triples	32 bits	120s
	64 bits	171s
Trusted bits	32 bits	32s
	64 bits	46s

they've computed the same hash value (Sec. 5.1). We use SHA-1 as the hash function. After the tuples have been generated, the prover signs the shares that the verifiers have.

To reduce the communication in step (ii) above, we let the prover share a common random seed with each of the verifiers. In this manner, the random values do not have to be sent. E.g. for a multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, both shares of $\llbracket a \rrbracket$, both shares of $\llbracket b \rrbracket$ and one share of $\llbracket c \rrbracket$ are random. The prover only has to send one of the shares of $\llbracket c \rrbracket$ to one of the verifiers.

Execution phase. A Sharemind computation server consists of several subsystems on top of each other. Central of those is the virtual machine (VM). This component reads the description of the privacy-preserving application and executes it. The description is stated in the form of a bytecode (compiled from a high-level language) which specifies the operations with public data, as well as the protocols to be called on private data. There is a large number (over 100) of compiled primitive protocols that may be called by the VM. These protocols are compiled from higher-level descriptions with one of the intermediate formats being very close to circuits in Fig. 1. The protocols call the networking methods in order to send a sequence of values to one of the other two computation servers, or to receive messages from them.

In order to support verification, a computation server of Sharemind must log the randomness the server is using, as well as the messages that it has sent or received. Using these logs, the descriptions of the privacy-preserving application and the primitive protocols, it is possible to restore the execution of the server.

We have modified the network layer of Sharemind, making it sign each message it sends, and verify the signature of each message it receives. We have not added the logic to detect whether two outgoing messages belong to the same round or not (in the former case, they could be signed together), but this would not have been necessary, because our compiled protocols produce only a single message for each round. We have used GNU Nettle for the cryptographic operations. For signing, we use 2 kbit RSA and SHA-256. Beside message signing and verification, we have also added the logging of all outgoing and incoming messages.

Verification phase. The virtual machine of the post-execution phase reads the application bytecode and the log of messages to learn, which protocols were invoked in which order and with which data during the execution phase. The information about invoked protocols is present in both the prover's log, as well as in the verifiers' logs. Indeed, the identity of invoked protocols depends only on the application, and on the public data it operates on. This is identical for all computation servers. The post-execution VM then reads the descriptions of protocols and performs the steps described in Sec. 4.7. The post-execution VM has been implemented in Java, translated with the OpenJDK 6 compiler and run in the OpenJDK 7 runtime environment. The verification phase requires parties to sign their messages, we have used 2 kbit RSA with SHA-1 for that purpose.

6.2 The Cost of Integer multiplication

We have measured the total cost of covertly secure private multiplication, using the tools that we have implemented. Our tests make use of three servers running on a LAN, similarly to the benchmarks reported in Sec. 2. We run a large number of 32-bit integer multiplications in parallel and report the execution time for a single multiplication.

Preprocessing. In the described set-up, we are able to generate 100 million verification triples for 32-bit multiplication in ca. 120 seconds (Table 1). To verify a single multiplication protocol, we need 9 such triples [11, Alg. 2]. Hence the amortized preprocessing effort to verify a single 32-bit multiplication is ca. 11µs.

Execution. Passively secure Sharemind can perform 22 million 32-bit multiplications per sec-

Table 2: Time and communication of the verification phase

# multiplications	Time (ms)					Communication (bits)
	prover local comp.	verifier local comp.	prover sign.	verifier sign. check	communication	
10^2	0.061	0.17	97	15	110	21696
10^3	0.73	1.2	118	24	240	194496
10^4	11	10	132	47	280	1922496
10^5	24	26	164	78	390	19202496
10^6	66	120	290	220	1100	192002496
10^7	270	450	1300	1300	7200	1920002496

ond [39], if a large number of them are computed in parallel on servers that support AES-NI. When message signing and logging have been added, the performance drops to 7 million multiplications per second, or 0.15µs per multiplication. In general, the signing and logging appears to reduce the performance of Sharemind approximately three times.

Verification. Assuming that all the inputs and the communication have already been committed, and the beaver triples precomputed, we run the verification phase and measure the time spent on the offline computation, the communication, and separately on signing (to see how much depends on the underlying signature scheme). We consider the optimistic setting, where the prover only signs the broadcast message, and the verifiers exchange the hash of the message to ensure that they got the same message. We also measure the number of total communicated bits. The results of verifying a single party are given in Table 2. For the whole verification effort, we have to add up the prover's execution time (2nd, 4th and 6th column), and twice the verifier's execution time (3rd, 5th and 6th column). When performing 10 million verifications in parallel, the cost to verify a single 32-bit multiplication is ca. 2.7µs.

When adding the costs of three phases, we find that the total amortized cost of performing a 32-bit multiplication in our three-party SMC protocol tolerating one actively corrupted party is ca. 14µs. This is more than two orders of magnitude faster than any existing solution.

6.3 Estimating the cost of other protocols

Our implementations of the preprocessing and verification phases are still preliminary, at least compared to the existing Sharemind platform and the engineering effort that has been gone into it. We believe that significant improvements in their running times are possible, even without changing the underlying algorithms or invoking extra protocol-level optimizations. Hence we are looking for another metric that may predict the running time of the new phases once they have been optimized. Due to the very simple communication pattern of that phase, consisting of the prover sending a large message to the verifiers, followed by the verifiers exchanging very small messages, we believe that the number of needed communication bits is a good proxy for future performance.

The existing descriptions of Sharemind's protocols make straightforward the computation of their execution and verification costs in terms of communicated bits. We have performed the computation for the protocols working with integers, and counted the number bits that need to be delivered for executing and verifying an instance of the protocol. We have not taken into account the signatures, the broadcast overhead, and the final alleged zero hashes that the verifiers exchange, because these can be amortized over a large number of protocols executing either in parallel or sequentially.

Table 3 presents our findings. For each protocol, the results are presented in the form $\begin{smallmatrix} x:y:z \\ 1:\boldsymbol{a}:b \end{smallmatrix}$. The upper line lists the total communication cost (in bits): x for the execution of the protocol, y for its verification in the post-execution phase, and z for the generation of precomputed tuples in the preprocessing phase. The suffixes k and M denote the multipliers 10^3 and 10^6, respectively. The lower line is computed straightforwardly from the upper line, and it shows how many times more expensive each phase is, compared to the execution phase (i.e $\boldsymbol{a} = y/x$, $b = z/x$). The most interesting value is $\boldsymbol{a}$ that shows how much overhead our verification gives in the online phase, compared to passively secure computation.

In estimating the costs of generating precomputed tuples, we have assumed the tuples to be generated in batches of 2^{20}, with security parameter $\eta = 80$. Sec. 4.4 describes the number of extra tuples that we must send for correctness checks. We consider the selected parameters rather conservative; we would need less extra tuples and less communication during the preprocessing phase if we increased the batch size or somewhat lowered the security parameter. Increasing the batch size to ca. 100 million would drop the parameter m from 5 to 4, thereby reducing the communication needs of preprocessing by 20%.

Table 3: Communication overheads of integer operation verification

Operation	bit width			
	8	16	32	64
multiplication	48 : 288 : 1513 *1* : ***6*** : *32*	96 : 576 : 3025 *1* : ***6*** : *32*	192 : 1152 : 6050 *1* : ***6*** : *32*	384 : 2304 : 12k *1* : ***6*** : *32*
division	4.4k : 81k : 4.9M *1* : ***18*** : *1120*	10k : 200k : 25M *1* : ***20*** : *2500*	32k : 690k : 190M *1* : ***21*** : *5900*	89k : 2.1M : 1400M *1* : ***23*** : *16000*
div. with pub.	452 : 12k : 320k *1* : ***26*** : *707*	1044 : 27k : 1.4M *1* : ***26*** : *1340*	2.4k : 64k : 6.2M *1* : ***26*** : *2600*	5.3k : 144k : 27M *1* : ***27*** : *5100*
priv. $\ll$ priv.	152 : 3.3k : 60k *1* : ***22*** : *390*	416 : 9.1k : 280k *1* : ***22*** : *670*	1328 : 35k : 2.1M *1* : ***26*** : *1580*	4.7k : 135k : 16M *1* : ***29*** : *3500*
priv. $\gg$ priv.	328 : 10k : 167k *1* : ***31*** : *510*	864 : 30k : 860k *1* : ***35*** : *990*	2.4k : 101k : 5.4M *1* : ***42*** : *2250*	7.2k : 360k : 37M *1* : ***49*** : *5100*
priv. $\gg$ pub.	204 : 4.6k : 74k *1* : ***22*** : *360*	500 : 11.5k : 330k *1* : ***23*** : *660*	1188 : 28k : 1.5M *1* : ***23*** : *1250*	2.8k : 66k : 6.6M *1* : ***24*** : *2400*
equality	50 : 804 : 7.2k *1* : ***16*** : *143*	106 : 1.8k : 25k *1* : ***17*** : *240*	218 : 3.8k : 94k *1* : ***18*** : *430*	442 : 7.9k : 360k *1* : ***18*** : *810*
less than	322 : 7.9k : 117k *1* : ***24*** : *360*	809 : 21k : 590k *1* : ***26*** : *730*	1.9k : 51k : 2.8M *1* : ***27*** : *1440*	4.5k : 121k : 13M *1* : ***27*** : *2900*
bit decomp.	184 : 3.8k : 60k *1* : ***21*** : *330*	464 : 10k : 278k *1* : ***22*** : *600*	1120 : 25k : 1.3M *1* : ***22*** : *1140*	2.6k : 60k : 5.9M *1* : ***23*** : *2230*

If we take $\eta = 40$, then $m = 3$ would be sufficient.

The described integer protocols in Table 3 take inputs additively shared between three computing parties and deliver similarly shared outputs. In the "standard" protocol set, the available protocols include multiplication, division (with private or with public divisor), bit shifts (with private or public shift), comparisons and bit decomposition, for certain bit widths. We left out the protocols for operations that require no communication between parties during execution or verification phase: addition, and multiplication with a constant.

We see that the verification overhead (normalized to communication during the execution phase) of different protocols varies quite significantly. While most of the protocols require 20–30 times more communication during the verification phase than in the execution phase, the important case of integer multiplication has the overhead of only six times. Even more varied are the overheads for preprocessing, with integer multiplication again having the smallest overhead of 31.5 and the protocols working on smaller data having generally smaller overheads.

Discussion

Our explanation to the variability of overheads is the following. We are measuring a parameter of the protocols that, up to now, has been considered completely irrelevant to their performance. The overheads of pre- and post-processing depend on the operations performed locally by the Sharemind servers during the execution phase. The overheads are particularly sensitive to the order of operations, and how similar are the consecutive operations performed with the "same" data. A n-bit value held by a server may be interpreted both as an element of $\mathbb{Z}_{2^n}$ or an element of $\mathbb{Z}_2^n$; the conversion is cost-free during the execution phase. During verification, such conversion requires us to do a bit-decomposition or a number of conversions to a larger ring. If the expressions evaluated during the execution contain a fine mix of arithmetic and bitwise operations, then the number of such conversion will be large and many trusted bits are consumed.

We have thus identified a new goal in optimizing SMC protocols, and Sharemind protocols in particular — the local computations of a server should be structured in a manner that minimizes the number of times a bitwise operation follows an arithmetic one or vice versa. Also, the number of operations that are not free to verify should be minimized in general. While we likely cannot achieve overheads as small as the multiplication protocol currently has (the servers perform no bitwise operations in this protocol, hence the issue of mixing operations does not arise), we hope that strategic placement of conversions

allows us to reduce the post-processing overhead a couple of times and preprocessing overhead by at least an order of magnitude. We tried to rewrite a small protocol — integer equality — to a form where arithmetic and bitwise operations were mostly separated from each other. The resulting protocol had the same cost in the execution phase, but its postprocessing overhead was only 4 times and the preprocessing overhead between 31 (for 8-bit inputs) and 112 times (for 64-bit inputs) of that.

Even the multiplication protocol can be further optimized for verification. In this protocol, each party performs a computation of the form $w = uv + u'v + uv'$ where u, v, u', v', w are ring elements; this is the only part of protocol where effort in verification has to be spent [11, Alg. 2], each party having to prove that it performed the three multiplications correctly. We may rewrite that expression to $w = (u+v)(u'+v') - u'v'$ containing only two multiplications and reducing the overheads by 1/3. In this manner, the *total communication cost* for performing multiplications secure against malicious adversaries would be only around 25 times larger than the effort of a passively secure protocol. With three servers on a LAN, Sharemind can currently compute around 22 million 32-bit multiplications per second [39], if these are massively parallelized; for 64-bit multiplications, the performance is half of that. Hence we believe that on a LAN, after optimizing the preprocessing and verification phases, we could perform several hundred thousand maliciously secure 32-bit multiplications per second, taking into account *all necessary pre- and post-processing*. This exceeds the earlier reported results, either based on linear secret sharing [21, 24] or garbled circuits [59], by almost three orders of magnitude.

One may ask whether the relatively higher cost of verifying other operations (besides multiplication) may diminish the advantages of our techniques over the state of the art when considering privacy-preserving applications that are more dependent in these other operations. This question may be answered both affirmatively and negatively. Closest to our performance are SPDZ-like protocols [24] built on top of additive secret sharing over fields $\mathbb{Z}_p$. These protocol sets do not "naturally" support many operations; instead, they have to build other private operations from the composition of multiplications and bit decompositions [20, 50]. Hence their performance is also worse for other operations. On the other hand, the protocol sets working with Boolean circuits (using either garbled circuits or secret sharing) do not pay similar performance penalty. But their currently discussed performance was another order of magnitude slower than for protocols based on sharing over $\mathbb{Z}_p$.

7 Conclusions and Further Work

We have proposed a scheme transforming passively secure protocols with honest majority to covertly secure ones. The protocol transformation is suitable to be implemented on top of some existing, highly efficient, passively secure SMC frameworks, especially those that use 3 parties and computation over rings of size 2^N. 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 increase only slightly. We evaluated the verification on top of the Sharemind SMC framework and found its overhead to be of acceptable size, roughly an order of magnitude larger than the complexity of the SMC protocols themselves included in the framework (which are already practicable). We note that the protocols of Sharemind are currently not optimized for verification, hence the overheads may become even smaller in the future.

The notion of verifiability that we achieve in this paper is very strong — a misbehaving party will remain undetected with only a negligible probability. The original notion of covert security [3] only required a malicious party to be caught with non-negligible probability. By randomly deciding (with probability p) after a protocol run whether it should be verified, our method still achieves covert security, but the *average* overhead of verification is reduced by $1/p$ times. It is likely that overheads smaller than the execution time of the original passively secure protocol may be achieved in this manner, while keeping the consequences of misbehaving sufficiently severe. Auditability (Sec. 5.4) helps in setting up the contractual environment that establishes the consequences.

We could use the verification procedure after each protocol round, thereby obtaining a fully actively secure SMC protocol. While the communication overhead of such solution would be the same, its total overhead will probably be larger than for the verification after the computation, because of a more complex communication pattern. Also, verification after the protocol run may allow further optimizations for such computations, where the effort to check its correctness is smaller than the effort to actually perform it [57]. Such optimizations are applicable if the original SMC protocol set preserves *privacy* (but not necessarily *correctness*) against active adversaries [52]. The extent of their applicability is a

subject of future work.

For three-party protocols, we see the combination of Sharemind's multiplication protocol with our verification mechanism as a suitable method for performing the precomputations of SPDZ-like SMC protocol sets. Even though we would in this manner only get security against a malicious minority, we still consider the outcome interesting, because the online phase of SPDZ is hard to beat even in this case. Also, the online phase would still be secure even against all-but-one malicious parties. There may be use cases where the number of corrupted parties increases between the precomputation phase and the actual protocol run. Again, the investigation of this use case, together with the optimizations to our scheme that may be possible due to working over fields, is a subject of future work.

References

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

[2] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 681–698. Springer, 2012.

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

[4] 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.

[5] Carsten Baum, Ivan Damgård, Tomas Toft, and Rasmus Zakarias. Better preprocessing for secure multiparty computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 327–345. Springer International Publishing, 2016.

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[7] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234. Springer, 2015.

[8] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Social Study Using Secure Computation. *Proceedings of Privacy Enhancing Technologies*, 2016, 2016. To appear in the 16th Privacy Enhancing Technologies Symposium.

[9] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In Alejandro Russo and Omer Tripp, editors, *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, page 53. ACM, 2014.

[10] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

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

[12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer, 2012.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012.

[14] 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.

[15] 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.

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

[17] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503. ACM, 2002.

[18] 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.

[19] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.

[20] 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.

[21] 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.

[22] 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.

[23] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263. Springer, 2012.

[24] 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.

[25] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [55], pages 643–662.

[26] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.

[27] Ivan Damgrd, Tomas Toft, and Rasmus Winther Zakarias. Fast multiparty multiplications from shared bits. Cryptology ePrint Archive, Report 2016/109, 2016. `http://eprint.iacr.org/`.

[28] Ernesto Damiani, Valerio Bellandi, Stelvio Cimato, Gabriele Gianini, Gerald Spindler, Matthis Grenzer, Niklas Heitmüller, and Philipp Schmechel. PRACTICE Deliverable D31.2: risk-aware deployment and intermediate report on status of legislative developments in data protection, October 2015. Available from `http://www.practice-project.eu`.

[29] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In Ray et al. [54], pages 1504–1517.

[30] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.

[31] 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.

[32] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A Unified Approach to MPC with Preprocessing Using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 711–735. Springer, 2015.

[33] 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.

[34] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In Burton S. Kaliski, Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197. Springer, 1997.

[35] 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.

[36] 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.

[37] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.

[38] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. `http://eprint.iacr.org/2016/505`, to appear in CCS 2016.

[39] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In Michael Brenner and Kurt Rohloff, editors, *Proceedings of WAHC'16 - 4th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2016.

[40] 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.

[41] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 285–300. USENIX Association, 2012.

[42] Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. In Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, volume 8783 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2014.

[43] Peeter Laud and Alisa Pankova. Verifiable Computation in Multiparty Protocols with Honest Majority. In Sherman S. M. Chow, Joseph K. Liu, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings*, volume 8782 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2014.

[44] Peeter Laud and Jaak Randmets. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1492–1503. ACM, 2015.

[45] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[46] Yehuda Lindell and Ben Riva. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In Ray et al. [54], pages 579–590.

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

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

[49] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [55], pages 681–700.

[50] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.

[51] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[52] Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 75–89. IEEE, 2015.

[53] Pille Pullonen. Actively secure two-party computation: Efficient Beaver triple generation. Master's thesis, University of Tartu, Aalto University, 2013.

[54] Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015.* ACM, 2015.

[55] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[56] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1989.

[57] Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

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

[59] Abhi Shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 523–534. ACM, 2013.

[60] Meril Vaht. The Analysis and Design of a Privacy-Preserving Survey System. Master's thesis, Institute of Computer Science, University of Tartu, 2015.