Preprocessing-Based Verification of Multiparty Protocols with Honest Majority

Roman Jagomägis[1], Peeter Laud[1], and Alisa Pankova[1,2,3]

[1] Cybernetica AS
[2] Software Technologies and Applications Competence Centre (STACC)
[3] University of Tartu
{peeter.laud|alisa.pankova|roman.jagomagis}@cyber.ee

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. In the preprocessing phase, each party generates and shares a sufficient amount of verified multiplication triples that will be later used to assist that party's proof. The execution phase, after which the computed result is already available to the parties, has only negligible overhead that comes from signatures on sent messages. In the postprocessing phase, the verifiers repeat the computation 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. 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 [36], 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 [10, 14, 20, 25], as well as various protocols optimized to solve concrete problems [13, 17, 19, 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 (see e.g. [48]). Achieving security against active adversaries may be expensive, hence intermediate classes (between passive and active) have been introduced.

In practical settings, it is often sufficient that the active adversary is detected not immediately after the malicious act, but at some point later. Hence ideas from verifiable computation (VC) [34] 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. All the inputs and the incoming/outgoing messages of the prover are secret-shared (using a threshold linear secret-sharing scheme) among all the other parties. 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 [5] (*Beaver triples*), which we adapt for verification. Before starting the verification (and even the execution), the prover generates and shares among the other parties sufficiently many such triples. Importantly, the prover provides a proof that these triples are generated and shared correctly. During verification, the correctness of precomputed triples implies the correctness of prover's computations.

The entire construction constitutes a variant of the GMW compiler [16, 36] from passively to actively secure protocols, showing that this technique can be highly efficient. 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 [10] employed in the Sharemind platform [9], demonstrating for the first time a method to achieve security against active adversaries for Sharemind. We note that the protocol set of Sharemind is very efficient [39], and its deployments [11, 37, 60] include the largest SMC applications ever [6, 7]. We discuss the difficulties with previous methods in Sec. 2.

From covert to active security. The verification step converts a protocol secure against a passive adversary to a protocol secure against *covert* adversary [2] 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.

In general, 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.

Moreover, the protocol set [10] is *private* against active adversaries, as long as no values are declassified [52]. If declassification is applied only to computation results at the end of the protocol, then prepending it with our verification step gives us an actively secure protocol [43].

Hence in this paper, our stated goal is to achieve a strong form of *covert security*, where any deviations from the protocol will remain undetected with only negligible probability. But we keep in mind that it is only a small step from this security property to fully active security.

Cost of precomputation. There exist other protocols for SMC secure against active adversaries (see Sec. 2) where the additional verification of the behaviour of parties causes only very modest overheads. Our verification phase, even while having a reasonable cost of its own, is not competitive with these approaches. However, the efficiency of the verification of these other approaches 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. An important difference of our triple generation from the other works is that the prover is allowed to know the values of the triples, which makes the triple generation significantly more efficient.

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 [40, 49], on additive sharing with MACs to check for correct behaviour [23, 25, 27], on Shamir's secret sharing [20, 58], and on the GMW protocol [36] 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 [5] 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 [12,23] or oblivious transfer [49]. For ensuring the correctness of tuples, the generation is followed by a much cheaper correctness check [23]. 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.

While these methods can be secure for dishonest majority, they lead to protocols that are in some sense weaker than ours — they do not allow the identification of a misbehaving party. Recently, an identification mechanism for SPDZ-like protocols has been proposed [59], but the complexity of determining the identity of a misbehaving party may be too high for being a sufficient deterrent.

For honest majority and three parties, a recent method [33] proposes the use of a highly efficient passively secure protocol for precomputing multiplication triples. Again, this method only allows the detection of misbehaviour, but no identification of the guilty party.

Previously, methods for post-execution verification of the correct behaviour of protocol participants have been presented in [3,21,42]. We note that the general outline of our verification scheme is similar to [3] — we both commit to certain values during protocol execution and perform computations with them afterwards. However, the committed 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 [9], which is based on additive sharing over finite rings among three *computing* parties. The number of parties providing inputs or receiving outputs may be much larger. Typically, the rings represent integers of certain length. 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.

Complexity of actively secure integer multiplication and AES

We are interested in bringing security against active adversaries to integer and floating-point operations, to be used in secure statistical analyses [7], scientific computations [37] or risk analysis [6]. 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. Their total execution time for a single AES circuit is around 80ms, when doing 1024 executions in parallel and using the security parameter $\eta = 80$ (bits). 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 [22], 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 [22]. In [23], 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 [20] (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 [26], 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 1Gbps LAN, with three parties, tolerating a single actively corrupted party, $\eta = 80$, negligible escape probability for a cheating adversary) in 15 µs. This is around two orders of magnitude faster than the performance reported above.

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.

Recently [24], amortized time $0.5\mu s$ was reported for computing a single AES block. However, it takes into account only the online phase. The authors do not provide benchmarks for preprocessing, but they estimate that using recent mechanisms for doing preprocessing, up to 10^5 AND gates could be computed per second. Assuming that a single AES block contains ca 6400 AND gates (as in our benchmarks), this would suffice for around 16 AES blocks per second, or $63ms$ per AES block. In this work, we compute a 128-bit AES block with covert security in $2.9ms$, including the preprocessing.

3 Ideal Functionality

Notation. Throughout this work, we use $\boldsymbol{x}$ to denote vectors, where x_i is the i-th coordinate of $\boldsymbol{x}$. 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 $(z_1, \ldots, z_{\lceil \log n_x \rceil}) = \mathsf{bd}(x)$ 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 Sharemind [10] 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. The protocol set of Sharemind also contains some other operations, and all of them can be handled by our verification method.

Execution Functionality. We specify our verifiable execution functionality in the universal composability (UC) framework [15]. 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

• **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 $\boldsymbol{r}_i$. For each $i \in \mathcal{C}$, it sends $(\mathsf{randomness}, i, \boldsymbol{r}_i)$ 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 $(\mathsf{input}, \boldsymbol{x}_i)$ 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 $\boldsymbol{m}_{ij}^{\ell}$. For all $j \in \mathcal{C}$, it sends $\boldsymbol{m}_{ij}^{\ell}$ to $\mathcal{A}_S$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\boldsymbol{m}_{ji}^{\ell}$ from $\mathcal{A}_S$.
• **After** r **rounds**, $\mathcal{F}_{vmpc}$ sends $(\mathsf{output}, \boldsymbol{m}_{1i}^{r}, \ldots, \boldsymbol{m}_{ni}^{r})$ 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 $\boldsymbol{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$.
$\mathcal{F}_{vmpc}$ defines $\mathcal{M} = \mathcal{B}_0 \cup \{i \in \mathcal{C} \mid \exists j \in [n], \ell \in [r'] : \boldsymbol{m}_{ij}^{\ell} \neq C_{ij}^{\ell}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}_{1i}^{1}, \ldots, \boldsymbol{m}_{ni}^{\ell-1})\}$.
• **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.

Fig. 1: The ideal functionality $\mathcal{F}_{vmpc}$ for verifiable computations

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 $\boldsymbol{m}_{ij}^{\ell}$ to the party $j \in [n]$ from the input $\boldsymbol{x}_i$, randomness $\boldsymbol{r}_i$ and the messages $\boldsymbol{m}_{j'i}^{k}$ $(k < \ell)$ that P_i has received before. All values $\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}_{ij}^{\ell}$ 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.

The aim of this paper is to construct an efficient protocol Π_{vmpc} and to prove the following theorem.

Theorem 1. *Let $\mathcal{C}$ be the set of corrupted parties. If $|\mathcal{C}| < n/2$ and there is a PKI that fixes the public keys of all parties, then Π_{vmpc} UC-emulates $\mathcal{F}_{vmpc}$.*

The following section gives the construction of the protocol and outlines its security proof. The full proof of security is deferred to App. C.

4 The Real Protocol

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) has 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. We show that this allows us to use linear 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 the deviator cannot be identified. We formalize this phase as a functionality $\mathcal{F}_{pre}$ given in Fig. 2.

$\mathcal{F}_{pre}$ 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 $(\mathsf{init}, \tilde{m}, \tilde{n}, \tilde{p})$ from each (honest) party, initialize triple and bit to empty arrays. Assign the functions $m \leftarrow \tilde{m}$, $n \leftarrow \tilde{n}$, $p \leftarrow \tilde{p}$.
Multiplication triple generation: On input (triple, id) from P_i, check if $\mathsf{triple}[id]$ exists. If it does, take $(\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)_{k\in[n]} \leftarrow \mathsf{triple}[id]$. Otherwise, generate $\boldsymbol{r}_x \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}^{n(id)}$, $\boldsymbol{r}_y \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}^{n(id)}$, and compute $\boldsymbol{r}_{xy} \leftarrow \boldsymbol{r}_x \cdot \boldsymbol{r}_y$. Compute the shares $(\boldsymbol{r}_x^k)_{k\in[n]}$, $(\boldsymbol{r}_y^k)_{k\in[n]}$, and $(\boldsymbol{r}_{xy}^k)_{k\in[n]}$ of $\boldsymbol{r}_x$, $\boldsymbol{r}_y$, and $\boldsymbol{r}_{xy}$ respectively. Assign $\mathsf{triple}[id] \leftarrow (\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)_{k\in[n]}$. If $p(id) \neq i$, send $(\boldsymbol{r}_x^i, \boldsymbol{r}_y^i, \boldsymbol{r}_{xy}^i)$ to P_i. Otherwise, send $(\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)_{k\in[n]}$ to P_i. For all $k \in \mathcal{C}$, send $(\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)$ also to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send all shares $(\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)_{k\in[n]}$ to $\mathcal{A}_S$.
Trusted bit generation: On input (bit, id) from P_i, check if $\mathsf{bit}[id]$ exists. If it does, take $(\boldsymbol{r}^k)_{k\in[n]} \leftarrow \mathsf{bit}[id]$. Otherwise, generate a vector of random bits $\boldsymbol{r} \overset{\$}{\leftarrow} \mathbb{Z}_2^{n(id)}$. Compute the shares $(\boldsymbol{r}^k)_{k\in[n]}$ of $\boldsymbol{r}$ over $\mathbb{Z}_{m(id)}^{n(id)}$. Assign $\mathsf{bit}[id] \leftarrow (\boldsymbol{r}^k)_{k\in[n]}$. Handle $(\boldsymbol{r}^k)_{k\in[n]}$ similarly to the multiplication triple shares.
Stopping: On input (stop) from $\mathcal{A}_S$, stop the functionality and output $\perp$ to all parties.

Fig. 2: Ideal functionality $\mathcal{F}_{pre}$

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. We formalize the verification phase as a functionality $\mathcal{F}_{verify}$ given in Fig. 3.

In the rest of this section, we describe the building blocks used by $\mathcal{F}_{pre}$ and $\mathcal{F}_{verify}$, and the protocols implementing them.

4.1 Building Blocks

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.

$\mathcal{F}_{verify}$ 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. 4.3), the input indices $\boldsymbol{xid}(id)$, and the output indices $\boldsymbol{yid}(id)$ 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 $(\mathsf{init}, \tilde{f}, \tilde{\boldsymbol{xid}}, \tilde{\boldsymbol{yid}}, \tilde{p}, \tilde{p'})$ from all the (honest) parties, initialize *comm* and *sent* to empty arrays. Assign the functions $f \leftarrow \tilde{f}$, $\boldsymbol{xid} \leftarrow \tilde{\boldsymbol{xid}}$, $\boldsymbol{yid} \leftarrow \tilde{\boldsymbol{yid}}$, $p \leftarrow \tilde{p}$, $p' \leftarrow \tilde{p}'$.
Input Commitment: On input $(\mathsf{commit\_input}, \boldsymbol{x}, id)$ from $P_{p(id)}$, and $(\mathsf{commit\_input}, id)$ from all honest parties, if $comm[id]$ exists, then do nothing. Otherwise, assign $comm[id] \leftarrow \boldsymbol{x}$. If $p(id) \in \mathcal{C}$, then $\boldsymbol{x}$ is chosen by $\mathcal{A}_S$.
Message Commitment: On input $(\mathsf{send\_msg}, \boldsymbol{m}, id)$ from $P_{p(id)}$, output $\boldsymbol{m}$ to $P_{p'(id)}$. If $p(id) \in \mathcal{C}$, then $\boldsymbol{m}$ is chosen by $\mathcal{A}_S$. If $p'(id) \in \mathcal{C}$, output $\boldsymbol{m}$ to $\mathcal{A}_S$. Assign $sent[id] \leftarrow \boldsymbol{m}$.
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 $comm[id] \leftarrow sent[id]$. If both $p(id), p'(id) \in \mathcal{C}$, assign $comm[id] \leftarrow \boldsymbol{m}^*$, where $\boldsymbol{m}^*$ 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, if $comm[id]$ exists. If it does, then do nothing. Otherwise, generate a random $\boldsymbol{r}$, and assign $comm[id] \leftarrow \boldsymbol{r}$. Output $\boldsymbol{r}$ to $P_{p(id)}$. If $p(id) \in \mathcal{C}$, output $\boldsymbol{r}$ also to $\mathcal{A}_S$.
Verification: On input (verify, id) from all honest parties, take $\boldsymbol{x} \leftarrow (comm[i])_{i \in \boldsymbol{xid}(id)}$ and $\boldsymbol{y} \leftarrow (comm[i])_{i \in \boldsymbol{yid}(id)}$. For $f \leftarrow f(id)$, compute $\boldsymbol{y'} \leftarrow f(\boldsymbol{x})$. If $\boldsymbol{y'} - \boldsymbol{y} = \boldsymbol{0}$, output 1 to each party. Otherwise, output $(\mathsf{corrupt}, k)$ to each party. Output the difference $\boldsymbol{y'} - \boldsymbol{y}$, to $\mathcal{A}_S$. Output the difference $\boldsymbol{y'} - \boldsymbol{y}$ to $\mathcal{A}_S$.
Stopping: On input (stop, k) from $\mathcal{A}_S$ for $k \in \mathcal{C}$, output $(\mathsf{corrupt}, k)$ to each party. Do not accept any inputs from $p(id) = k$ anymore. Assign $p(id) \leftarrow p'(id)$ for $p(id) = k$, and $p'(id) \leftarrow p(id)$ for $p'(id) = k$ (in order to make the proofs of the other parties possible to finish).

Fig. 3: Ideal functionality $\mathcal{F}_{verify}$

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 [21]. If the receiver claims that the sender has not sent the message, then the sender has to broadcast the message, so that each party is now convinced that the message has been delivered. If the sender refuses to broadcast, it will be publicly blamed. In the optimistic setting, as far as all the parties follow the protocol, this broadcast will not be needed at all. 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.

Sharing Based Commitments The commitment of the inputs, the randomness, the messages, and also the precomputed values, is based on a linearly homomorphic (n, t)-threshold sharing scheme with signatures. It ensures consistency of the shared value and allows to prove later what exactly 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, including the ones that do not 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. An (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 $\boldsymbol{a}^k \in R^{\binom{n}{t}}$, such that for each $j \in [\binom{n}{t}]$, the equation $\sum_{k \in \mathcal{V}_j} a^k_j = a$ holds. Also, $a^k_j = 0$ 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 $[\![a]\!] = (\boldsymbol{a}^k)_{k\in[n]}$ to denote the sharing of a.

Under the honest majority assumption, any (n,t)-threshold secret sharing scheme with $t = \lfloor n/2 \rfloor + 1$ can be used as a commitment [18]. Namely, the committed value is shared among the n parties, and each share is signed (we assume the availability of a PKI). The commitment can be opened by revealing 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 failure to open the commitment). 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.

4.2 Protocol Implementing $\mathcal{F}_{pre}$

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 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, the parties will not run the execution phase.

In order 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 the cut-and-choose step, the parties randomly select k tuples and open them. This phase fails if any of the opened tuples are wrong. If cut-and-choose succeeds, only a negligible amount of the remaining tuples may be wrong. The parties then 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. The preceding cut-and-choose step ensures that all the finally used tuples are correct, except with negligible probability.

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, and its results are quite similar to [33]) 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. For example, we can replace the cut-and-choose and pairwise verification steps with an application of linear error correcting codes [4]. 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.

The discussion of this section can be seen as a proof sketch for the following lemma:

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 $\mathcal{F}_{pre}$.*

The cut-and-choose with pairwise verification works similarly to [25] and [33], so we refer to [33] for a more formal proof of Lemma 1.

4.3 Protocol Implementing $\mathcal{F}_{verify}$

The initialization of $\mathcal{F}_{verify}$ triggers initialization of $\mathcal{F}_{pre}$ that reserves the identifiers and the corresponding types of tuples that will be later used to verify the functions $f(id)$ with which $\mathcal{F}_{verify}$ is initialized.

One by one, we now describe how different functionalities of $\mathcal{F}_{verify}$ are implemented.

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 $\boldsymbol{r}_j$ to the prover P and commits both herself and the prover to it. More precisely, V_j secret-shares the vector $\boldsymbol{r}_j$, signs the shares, and sends them all to P. The prover P checks for the correctness of sharing and the signatures, and forwards a signed share to each verifier. The prover P uses $\sum_j \boldsymbol{r}_j$ as its randomness. The sharing of this sum can be computed as the sum of the shares of all $\boldsymbol{r}_j$.

This method works as far as the protocol rounds are synchronous, so that a corrupted prover cannot collaborate with some corrupted verifier to tamper with the randomness share *after* all the shares of the honest verifiers get known to P. In this particular case, synchronicity can be enforced by requiring each verifier to send first to each other party the hash of the shares that it will obtain from P after one round, and only then transmit to P the actual shares. In this way, the prover is bound to the t shares of each $\boldsymbol{r}_j$ issued to the honest parties already before any $\boldsymbol{r}_j$ is revealed to P. We get a subprotocol implementing commit_rnd.

Commitments to inputs and messages Before the execution phase starts, the prover P commits to its input x by secret-sharing it. This implements commit_input.

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 [35] may be useful. This implements the function send_msg that is a prerequisite for commit_msg.

After the execution phase ends, 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. If the sharing was incorrect, P' publishes the message, its shares, and all P's signatures, demonstrating that P has misbehaved. 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. In the latter case, publishing m does not leak anything, since if at least one of P and P' is corrupted, then the adversary has already seen m.

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. It may happen that the sharing $[\![m]\!]$ does not correspond to the 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.

We get a subprotocol implementing $\mathsf{commit\_msg}$.

Verification Verification of basic operations. A circuit (defined in Sec. 3) is composed of addition, multiplication, bit decomposition (bd), 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) one round later.

Verifying linear combinations. The verifiers compute locally the linear combination of 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 $\hat{x}_2 = x_2 - b$. For an honest prover, this is the first time the values a and b are used, and since these values are uniformly distributed over $\mathbb{Z}_n$, they serve as masks for x_1 and x_2.

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 $[\![z]\!] = [\![x]\!] - \sum_{i=0}^{n-1} 2^i \cdot [\![y_i]\!]$. Similarly to the multiplication case, this is the first time trusted bits b_i are used, and hence c_i are distributed uniformly over {0,1} in $\mathbb{Z}_{2^n}$.

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 $[\![z]\!] = [\![x]\!] - \sum_{i=1}^{n-1} 2^i \cdot \mathsf{trunc}([\![y'_i]\!])$. All the published values are distributed uniformly, by reasoning similar to the previous cases.

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.

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 broadcasts 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 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. 4.3, 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.

Putting the subprotocols of $\mathcal{F}_{verify}$ together Putting together the subprotocols of Sec. 4.3, and assuming identifiable abort (ensured by the message transmission of Sec. 4.1 that we use for sending all messages), we get a method that allows to blame any party that deviated from the protocol in any of its three phases. It may be less clear at this point whether the real protocol allows blaming more parties, that have not misbehaved during the execution phase.

The function stop of $\mathcal{F}_{verify}$ allows to blame any party P_k for $k \in \mathcal{C}$, so it is not a problem if any corrupted party is blamed due to deviating from the additional verification-related steps. Note that, in our implementation of $\mathcal{F}_{verify}$, the accusations may take place in the following two cases:

1. The party P_i failed in proving its honestness in the execution phase. If P_i is honest, then it has honestly followed the execution phase, and it always committed only the values that lead to successful alleged zero check. Under honest majority assumption, using threshold secret sharing with signatures ensures that the corrupted parties cannot tamper with their shares and force the proofs of P_i to fail.
2. Some other party presents a signature, proving that P_i has generated a set of shares that does not correspond to the previously signed message (during commitments), or that P_i generated a share that does not correspond to the shares it has committed before (during the verification). An honest party never signs a value that does not correspond to its previous signatures.

The discussion above gives us a proof sketch for the following lemma.

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.*

Using $\mathcal{F}_{verify}$ for commitments of all the input, randomness, and communication; and to later verify the computation on this values, we get an implementation of $\mathcal{F}_{vmpc}$, thus proving Theorem. 1.

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 $[\![a]\!] = (\boldsymbol{a}^k)_{k\in[n]}$ we can delete from the vectors $\boldsymbol{a}^k$ (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. 4.1. In particular, if $n = 3$ (as in Sharemind), then $[\![a]\!] = (a_1, a_2)$, where $a_1 + a_2 = a$

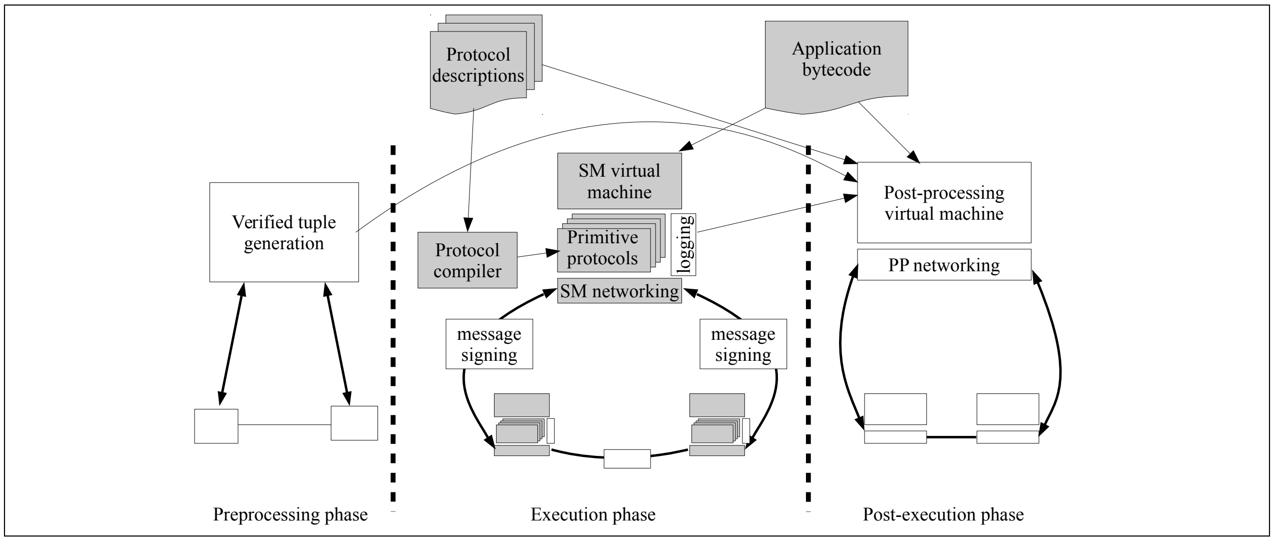

Fig. 4: Components of Sharemind with verification

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 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 [10, 39, 41, 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 of how messages are computed and exchanged between parties [44]. The application itself is described in a high-level language that is compiled into bytecode [8], 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. 4.2); (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 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. 4.3. The post-execution VM has

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	217 s
	64 bits	313 s
Trusted bits	32 bits	68 s
	64 bits	93 s
xor-shared AND triples	32 bits	217 s

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 Total Cost of Covertly Secure Protocols

For benchmarking, we have chosen the most general protocols of Sharemind over $\mathbb{Z}_{2^{32}}$: bitwise conjunction (AND32), multiplication (MULT32), 128-bit AES (AES128), conversion from additive sharing (i.e. over $\mathbb{Z}_{2^{32}}$) to xor sharing (i.e. over $\mathbb{Z}_2^{32}$) (A2X32) and vice versa (X2A32). We have measured the total cost of covert security of these protocols, using the tools that we have implemented. Our tests make use of three 2× Intel Xeon E5-2640 v3 2.6 GHz/8GT/20M servers, with 125GB RAM running on a 1Gbps LAN, similarly to the benchmarks reported in Sec. 2. We run a large number of protocol instances in parallel, and report the amortized execution time for a single protocol.

Preprocessing. In the described set-up, we are able to generate 100 million verification triples for 32-bit multiplication in ca. 217 seconds (Table 1). To verify a single multiplication protocol, we need 6 such triples (we use Sharemind protocol [10, Alg. 2] that formally has 3 multiplications per party, but all of them are of the form $x_1y_1 + x_1y_2 + x_2y_1$ and can be trivially rewritten to $x_1(y_1 + y_2) + x_2y_1$). Hence the amortized preprocessing effort to verify a single 32-bit multiplication is ca. 13 µs.

Sharemind uses 6400 AND gates per AES128 block. Each AND gate is just a multiplication, and it requires 6 one-bit triples. The time of generating 10^8 xor-shared 32-bit AND triples is 217 s. Hence the amortized preprocessing effort to verify a single 128-bit AES block is ca. 2.6 ms.

The A2X [resp. X2A] protocol requires 96 xor-shared AND triples [resp. 64 additively shared 32-bit multiplication triples], and 64 [resp. 96] 32-bit trusted bits. The amortized effort of these protocols is ca. 252 µs for A2X, and 204 µs for X2A.

Execution. We have measured runtimes of passively secure Sharemind with and without signing and logging. The execution times in milliseconds are given in Table 2. If a large number of these operations are computed in parallel, the amortized time (including all necessary signing and logging) is ca 0.16 µs for AND32 and MULT32, 0.04 ms for one AES128 block, 2.3 µs for A2X, and 5.1 µs for X2A. In general, for sufficiently large inputs, the signing and logging appears to reduce the performance of the current implementation of Sharemind up to three times. It is likely that a more careful parallelization of the networking layer would eliminate most of that overhead, by performing the signature creations and verifications on currently idling processor cores.

Verification. Assuming that all the inputs and the communication have already been committed, and the beaver triples precomputed, we run the verification phase in parallel for all 3 provers, and measure the total execution time (for asymmetric protocols, we report the times of all 3 provers). 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. The results are given in Table 3. When performing 10 million verifications in parallel, the cost of verification is ca. 1.6 µs for MULT32 (or AND32), 0.27 ms for a single AES128 block, 21 µs for A2X32, and 39 µs for X2A32.

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 covertly corrupted party is ca. 15 µs. This is more than two orders of magnitude faster than any existing solution. For a single AND gate, we get 0.46 µs.

Table 2. Times of the execution phase with and without signing and logging (ms)

# runs	AND32		MULT32		AES128		A2X32		X2A32	
	w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
10^1	0.362	4.75	0.349	3.96	11.3	485	0.785	38.8	0.19	8.75
10^2	0.345	4.42	0.237	3.84	13.4	496	0.928	38.7	1.05	8.59
10^3	0.147	4.58	0.282	4.04	33.0	600	1.73	45.0	2.28	12.8
10^4	0.668	6.37	0.733	5.40	214	726	8.44	55.6	27.3	60.4
10^5	7.46	15.1	8.13	15.1	2090	3740	98.4	227	252	481
10^6	73.9	166	73.8	184	–	–	909	2290	2690	5050
10^7	683	1550	717	1630	–	–	–	–	–	–

Table 3. Time and communication of the verification phase

# runs	time (s)								
	AND32	MULT32	AES128	A2X32			X2A32		
				P1	P2	P3	P1	P2	P3
10^1	0.316	0.319	0.434	0.315	0.329	0.317	0.329	0.329	0.322
10^2	0.317	0.320	0.678	0.369	0.362	0.356	0.362	0.397	0.401
10^3	0.371	0.373	1.11	0.471	0.452	0.482	0.452	0.516	0.533
10^4	0.558	0.569	4.21	0.891	0.829	0.911	0.829	0.109	1.12
10^5	0.895	0.906	2.67	2.55	2.43	2.93	2.43	0.491	5.50
10^6	2.48	2.50	–	17.2	15.8	20.4	15.8	34.3	39.1
10^7	15.5	15.5	–	–	–	–	–	–	–

The total cost of evaluating a 128-bit AES block is ca 2.9 ms, which is at least one order of magnitude faster than the existing solutions. The total cost of conversions between additive and bitwise sharing is ca. 275 µs for A2X32 and 248 µs for X2A32, and we could not find similar results in related works, with whom we could compare ourselves.

The recent result of computing AND gates [33] does not report times, but uses total number of communicated bits per AND gate instead. Their reported number is 30 bits per AND gate for 3 parties. Using the same security parameter $\eta = 40$ (taking $m = 3$), and making use of shared randomness, we get that the generation one 1-bit multiplication triple requires 1 bit of communication and each pairwise verification 4 bits (opening the 2 masked values by 2 verifiers to each other), adding up to $1 + 4 \cdot (m - 1) = 9$ bits for a single verified triple. Since we require a triple for each of the 6 local multiplications of Sharemind protocol, we already get 54 bits. The execution phase requires 6 bits of communication, and the verification phase 24 bits (8 for each party). This is 84 bits in total, or almost three times more. Some additional overhead may come from signatures (their cost becomes negligible as the communication grows). But our security property is stronger, allowing to pinpoint the cheating party and make the protocol aborting identifiable. Our method is also more generic and allows to easily generate preprocessed tuples other than multiplication triples, that are very useful in verifying protocols other than multiplication.

Instead of Sharemind multiplication, we could apply our verification to the protocol of [1]. This would reduce the execution phase time, but complexities of the preprocessing and the verification phases remain the same. It is not clear how well that protocol could be integrated with the other Sharemind protocols, so it becomes more interesting when more various protocols composable with [1] will be developed.

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).

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 [2] 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. 5.2) 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 (or the protocol of [1]) 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. Araki, T., Furukawa, J., Lindell, Y., Nof, A., and Ohara, K. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 805–817.
2. Aumann, Y., and Lindell, Y. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology 23*, 2 (2010), 281–343.
3. Baum, C., Damgård, I., and Orlandi, C. Publicly auditable secure multi-party computation. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings* (2014), M. Abdalla and R. D. Prisco, Eds., vol. 8642 of *Lecture Notes in Computer Science*, Springer, pp. 175–196.
4. Baum, C., Damgård, I., Toft, T., and Zakarias, R. Better preprocessing for secure multiparty computation. In *Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings* (2016), M. Manulis, A.-R. Sadeghi, and S. Schneider, Eds., Springer International Publishing, pp. 327–345.
5. Beaver, D. Efficient multiparty protocols using circuit randomization. In *CRYPTO* (1991), J. Feigenbaum, Ed., vol. 576 of *Lecture Notes in Computer Science*, Springer, pp. 420–432.
6. Bogdanov, D., Jõemets, M., Siim, S., and Vaht, M. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers* (2015), R. Böhme and T. Okamoto, Eds., vol. 8975 of *Lecture Notes in Computer Science*, Springer, pp. 227–234.
7. Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., and Talviste, R. 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.

8. Bogdanov, D., Laud, P., and Randmets, J. Domain-polymorphic programming of privacy-preserving applications. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014* (2014), A. Russo and O. Tripp, Eds., ACM, p. 53.
9. Bogdanov, D., Laur, S., and Willemson, J. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS* (2008), S. Jajodia and J. López, Eds., vol. 5283 of *Lecture Notes in Computer Science*, Springer, pp. 192–206.
10. Bogdanov, D., Niitsoo, M., Toft, T., and Willemson, J. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec. 11*, 6 (2012), 403–418.
11. Bogdanov, D., Talviste, R., and Willemson, J. Deploying secure multi-party computation for financial data analysis - (short paper). In *Financial Cryptography* (2012), A. D. Keromytis, Ed., vol. 7397 of *Lecture Notes in Computer Science*, Springer, pp. 57–64.
12. Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012* (2012), S. Goldwasser, Ed., ACM, pp. 309–325.
13. Brickell, J., and Shmatikov, V. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT* (2005), B. K. Roy, Ed., vol. 3788 of *Lecture Notes in Computer Science*, Springer, pp. 236–252.
14. Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium* (Washington, DC, USA, 2010), pp. 223–239.
15. Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS* (2001), IEEE Computer Society, pp. 136–145.
16. Canetti, R., Lindell, Y., Ostrovsky, R., and Sahai, A. Universally composable two-party and multi-party secure computation. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada* (2002), J. H. Reif, Ed., ACM, pp. 494–503.
17. Catrina, O., and de Hoogh, S. Secure multiparty linear programming using fixed-point arithmetic. In *ESORICS* (2010), D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., vol. 6345 of *Lecture Notes in Computer Science*, Springer, pp. 134–150.
18. Cramer, R., Damgård, I., and Maurer, U. M. General secure multi-party computation from any linear secret-sharing scheme. In *EUROCRYPT* (2000), B. Preneel, Ed., vol. 1807 of *Lecture Notes in Computer Science*, Springer, pp. 316–334.
19. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J. B., and Toft, T. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC* (2006), S. Halevi and T. Rabin, Eds., vol. 3876 of *Lecture Notes in Computer Science*, Springer, pp. 285–304.
20. Damgård, I., Geisler, M., Krøigaard, M., and Nielsen, J. B. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography* (2009), S. Jarecki and G. Tsudik, Eds., vol. 5443 of *Lecture Notes in Computer Science*, Springer, pp. 160–179.
21. Damgård, I., Geisler, M., and Nielsen, J. B. From passive to covert security at low cost. In *TCC* (2010), D. Micciancio, Ed., vol. 5978 of *Lecture Notes in Computer Science*, Springer, pp. 128–145.
22. Damgård, I., Keller, M., Larraia, E., Miles, C., and Smart, N. P. Implementing AES via an actively/-covertly secure dishonest-majority MPC protocol. In *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings* (2012), I. Visconti and R. D. Prisco, Eds., vol. 7485 of *Lecture Notes in Computer Science*, Springer, pp. 241–263.
23. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS* (2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.
24. Damgård, I., Nielsen, J. B., Nielsen, M., and Ranellucci, S. Gate-scrambling revisited - or: The tinytable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695, 2016. `http://eprint.iacr.org/2016/695`.
25. Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [55], pp. 643–662.
26. Damgård, I., Toft, T., and Zakarias, R. W. Fast multiparty multiplications from shared bits. Cryptology ePrint Archive, Report 2016/109, 2016. `http://eprint.iacr.org/`.
27. Damgård, I., and Zakarias, S. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC* (2013), pp. 621–641.
28. Damiani, E., Bellandi, V., Cimato, S., Gianini, G., Spindler, G., Grenzer, M., Heitmüller, N., and Schmechel, P. 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. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., and Zeitouni, S. Automated synthesis of optimized circuits for secure computation. In Ray et al. [54], pp. 1504–1517.
30. Demmler, D., Schneider, T., and Zohner, M. 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* (2015), The Internet Society.
31. Franklin, M. K., Gondree, M., and Mohassel, P. Communication-efficient private protocols for longest common subsequence. In *CT-RSA* (2009), M. Fischlin, Ed., vol. 5473 of *Lecture Notes in Computer Science*, Springer, pp. 265–278.
32. Frederiksen, T. K., Keller, M., Orsini, E., and Scholl, P. A Unified Approach to MPC with Preprocessing Using OT. In *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* (2015), T. Iwata and J. H. Cheon, Eds., vol. 9452 of *Lecture Notes in Computer Science*, Springer, pp. 711–735.
33. Furukawa, J., Lindell, Y., Nof, A., and Weinstein, O. High-throughput secure three-party computation for malicious adversaries and an honest majority. Cryptology ePrint Archive, Report 2016/944, 2016. `http://eprint.iacr.org/2016/944`.
34. Gennaro, R., Gentry, C., and Parno, B. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO* (2010), T. Rabin, Ed., vol. 6223 of *Lecture Notes in Computer Science*, Springer, pp. 465–482.
35. Gennaro, R., and Rohatgi, P. How to sign digital streams. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings* (1997), B. S. Kaliski, Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 180–197.
36. Goldreich, O., Micali, S., and Wigderson, A. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC* (1987), ACM, pp. 218–229.
37. Kamm, L., and Willemson, J. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security* (2014), 1–18.
38. Keller, M., Orsini, E., and Scholl, P. 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. Kerik, L., Laud, P., and Randmets, J. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *Proceedings of WAHC'16 - 4th Workshop on Encrypted Computing and Applied Homomorphic Cryptography* (2016), M. Brenner and K. Rohloff, Eds.
40. Kreuter, B., Shelat, A., and Shen, C. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012* (2012), T. Kohno, Ed., USENIX Association, pp. 285–300.
41. Krips, T., and Willemson, J. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings* (2014), S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. Yiu, Eds., vol. 8783 of *Lecture Notes in Computer Science*, Springer, pp. 179–197.
42. Laud, P., and Pankova, A. Verifiable Computation in Multiparty Protocols with Honest Majority. In *Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings* (2014), S. S. M. Chow, J. K. Liu, L. C. K. Hui, and S. Yiu, Eds., vol. 8782 of *Lecture Notes in Computer Science*, Springer, pp. 146–161.
43. Laud, P., and Pettai, M. Secure multiparty sorting protocols with covert privacy. In *Proceedings of Nordsec 2016* (2016).
44. Laud, P., and Randmets, J. 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* (2015), ACM, pp. 1492–1503.
45. Laur, S., Willemson, J., and Zhang, B. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11* (2011), pp. 262–277.
46. Lindell, Y., and Riva, B. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In Ray et al. [54], pp. 579–590.
47. Merkle, R. C. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.
48. Mohassel, P., Orobets, O., and Riva, B. Efficient Server-Aided 2PC for Mobile Phones. *Proceedings of Privacy Enhancing Technologies 2016*, 2 (2016), 82–99.
49. Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [55], pp. 681–700.

50. Nishide, T., and Ohta, K. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography* (2007), T. Okamoto and X. Wang, Eds., vol. 4450 of *Lecture Notes in Computer Science*, Springer, pp. 343–360.
51. Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999), pp. 223–238.
52. Pettai, M., and Laud, P. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015* (2015), C. Fournet, M. W. Hicks, and L. Viganò, Eds., IEEE, pp. 75–89.
53. Pullonen, P. Actively secure two-party computation: Efficient Beaver triple generation. Master's thesis, University of Tartu, Aalto University, 2013.
54. Ray, I., Li, N., and Kruegel, C., Eds. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), ACM.
55. Safavi-Naini, R., and Canetti, R., Eds. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), vol. 7417 of *Lecture Notes in Computer Science*, Springer.
56. Schnorr, C. Efficient identification and signatures for smart cards. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings* (1989), G. Brassard, Ed., vol. 435 of *Lecture Notes in Computer Science*, Springer, pp. 239–252.
57. Schoenmakers, B., and Veeningen, M. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers* (2015), T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, Eds., vol. 9092 of *Lecture Notes in Computer Science*, Springer, pp. 3–22.
58. Shamir, A. How to share a secret. *Commun. ACM 22*, 11 (1979), 612–613.
59. Spini, G., and Fehr, S. Cheater detection in SPDZ multiparty computation. In *Information Theoretic Security - 9th International Conference, ICITS 2016, Tacoma, WA, USA, August 9-12, 2016, Revised Selected Papers* (2016), A. C. A. Nascimento and P. Barreto, Eds., vol. 10015 of *Lecture Notes in Computer Science*, pp. 151–176.
60. Vaht, M. The Analysis and Design of a Privacy-Preserving Survey System. Master's thesis, Institute of Computer Science, University of Tartu, 2015.

A Other operations

The circuits for computing the messages in certain protocols of Sharemind use some more operations in addition to those described in Sec. 4.3. We now describe their verification. Note that the multiplication protocol only needs multiplications to be verified [10, 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 $([\![r]\!], [\![\boldsymbol{s}]\!])$, 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 $\boldsymbol{s}$ is n, and $s_i = 1$ iff $i = r$. The prover broadcasts $\hat{x} = r - x' \in \mathbb{Z}_n$. The verifiers compute $[\![\boldsymbol{s}']\!] = \mathsf{rot}(\hat{x}, [\![\boldsymbol{s}]\!])$, 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. 4.2, with the following pairwise verification operation. Given tuples $([\![r]\!], [\![\boldsymbol{s}]\!])$ and $([\![r']\!], [\![\boldsymbol{s}']\!])$, the verifiers compute $[\![\hat{r}]\!] = [\![r']\!] - [\![r]\!]$, declassify it, compute $[\![\hat{\boldsymbol{s}}]\!] = [\![\boldsymbol{s}]\!] - \mathsf{rot}(\hat{r}, [\![\boldsymbol{s}']\!])$, declassify it and check that it is a vector of zeroes. Recall (Sec. 4.2) 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 $[\![\boldsymbol{y}]\!] = \mathsf{rot}([\![x']\!], [\![\boldsymbol{x}]\!])$ for $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!] \in \mathbb{Z}_{2^n}^m$ 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* $([\![r]\!], [\![\boldsymbol{s}]\!], [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$, where $[\![r]\!]$ and $[\![\boldsymbol{s}]\!]$ are a CV tuple (with $r \in \mathbb{Z}_m$ and $\boldsymbol{s} \in \mathbb{Z}_{2^n}^m$), $\boldsymbol{a} \in \mathbb{Z}_{2^n}^m$ is random and the elements of $\boldsymbol{b}$ satisfy $b_i = a_{(i+r) \bmod m}$.

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

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

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

The pairwise verification of rotation tuples
$\mathbf{T} = ([\![r]\!], [\![\boldsymbol{s}]\!], [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$ and $\mathbf{T}' = ([\![r']\!], [\![\boldsymbol{s}']\!], [\![\boldsymbol{a}']\!], [\![\boldsymbol{b}']\!])$ works similarly, using the tuple $\mathbf{T}'$ to rotate $[\![\boldsymbol{a}]\!]$ by $[\![r]\!]$ positions and checking that the result is equal to $[\![\boldsymbol{b}]\!]$ (i.e. subtract one from another, open and check that the outcome is a vector of zeroes). Additionally, pairwise verification of CV tuples is performed on $([\![r]\!], [\![\boldsymbol{s}]\!])$ and $([\![r']\!], [\![\boldsymbol{s}']\!])$.

Shuffle. The parties want to apply a permutation σ to a vector $[\![\boldsymbol{x}]\!] \in \mathbb{Z}_n^m$, obtaining $[\![\boldsymbol{y}]\!]$ 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*
$([\rho], [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$, where $\rho \in S_m$, $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_n^m$ and $\boldsymbol{b} = \rho(\boldsymbol{a})$. 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 $\hat{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{a}$. The verifiers compute their shares $(\boldsymbol{y}_1, \boldsymbol{y}_2)$ of $[\![\boldsymbol{y}]\!]$ as $\boldsymbol{y}_1 = \tau(\boldsymbol{b}_1 + \rho(\hat{\boldsymbol{x}}))$ and $\boldsymbol{y}_2 = \tau(\boldsymbol{b}_2)$, where $\boldsymbol{b}_i$ is the i-th verifier's share of $[\![\boldsymbol{b}]\!]$. The alleged zeroes $[\![\boldsymbol{z}]\!] = [\![\boldsymbol{x}]\!] - [\![\boldsymbol{a}]\!] - \hat{\boldsymbol{x}}$ are produced.

The pairwise verification of permutation triples $([\rho], [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$ and $([\rho'], [\![\boldsymbol{a}']\!], [\![\boldsymbol{b}']\!])$ again works similarly, using the second tuple to apply $[\rho]$ to $[\![\boldsymbol{a}']\!]$. The result is then checked for its equality to $[\![\boldsymbol{b}']\!]$.

B Estimating the cost of other Sharemind 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 4 presents our findings. For each protocol, the results are presented in the form $\frac{x:y:z}{1:a:b}$. 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. 4.2 describes the number of extra tuples that we must

Table 4. Communication overheads of integer operation verification

Operation	bit width			
	8	16	32	64
multiplication	48 : 192 : 1008 *1 :* ***4*** *: 21*	96 : 384 : 2017 *1 :* ***4*** *: 21*	192 : 768 : 4034 *1 :* ***4*** *: 21*	384 : 1536 : 8067 *1 :* ***4*** *: 21*
division	4178 : 46.0k : 1.1M *1 :* ***10*** *: 272*	9752 : 106.5k : 5.0M *1 :* ***10*** *: 514*	31.2k : 339.6k : 28.5M *1 :* ***10*** *: 914*	87.6k : 941.4k : 181.2M *1 :* ***10*** *: 2069*
div. with pub.	404 : 4812 : 94.4k *1 :* ***11*** *: 234*	948 : 11.3k : 339.9k *1 :* ***11*** *: 359*	2180 : 26.1k : 1.3M *1 :* ***11*** *: 581*	4932 : 59.1k : 4.8M *1 :* ***11*** *: 982*
priv. $\ll$ priv.	144 : 1472 : 20.7k *1 :* ***10*** *: 144*	400 : 5504 : 141.3k *1 :* ***13*** *: 353*	1296 : 21.2k : 1.1M *1 :* ***16*** *: 811*	4624 : 83.5k : 8.1M *1 :* ***18*** *: 1758*
priv. $\gg$ priv.	328 : 4592 : 35.8k *1 :* ***14*** *: 109*	864 : 16.9k : 185.9k *1 :* ***19*** *: 215*	2352 : 52.9k : 314.0k *1 :* ***22*** *: 134*	7120 : 198.8k : 1.1M *1 :* ***27*** *: 161*
priv. $\gg$ pub.	180 : 1626 : 14.8k *1 :* ***9*** *: 82*	468 : 4090 : 52.9k *1 :* ***8*** *: 113*	1092 : 9690 : 182.8k *1 :* ***8*** *: 167*	2564 : 22.4k : 658.2k *1 :* ***8*** *: 257*
equality	50 : 200 : 1571 *1 :* ***4*** *: 31*	106 : 424 : 4549 *1 :* ***4*** *: 43*	218 : 872 : 14.3k *1 :* ***4*** *: 66*	442 : 1768 : 49.3k *1 :* ***4*** *: 112*
less than	280 : 2748 : 16.0k *1 :* ***9*** *: 57*	719 : 7440 : 46.0k *1 :* ***10*** *: 64*	1750 : 18.7k : 127.3k *1 :* ***10*** *: 73*	4109 : 44.7k : 354.7k *1 :* ***10*** *: 86*
additive to xor	160 : 1120 : 6403 *1 :* ***7*** *: 40*	416 : 3008 : 18.1k *1 :* ***7*** *: 44*	1024 : 7552 : 49.4k *1 :* ***7*** *: 48*	2432 : 18.2k : 135.5k *1 :* ***7*** *: 56*
xor to additive	80 : 560 : 3722 *1 :* ***7*** *: 47*	288 : 2144 : 14.7k *1 :* ***7*** *: 51*	1088 : 8384 : 58.7k *1 :* ***7*** *: 54*	4224 : 33.2k : 234.2k *1 :* ***7*** *: 55*

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

The described integer protocols in Table 4 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 7–20 times more communication during the verification phase than in the execution phase, the important case of integer multiplication has the overhead of only four times. Even more varied are the overheads for preprocessing, with integer multiplication again having the smallest overhead of 21 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. We have already tried to optimize the local computations of protocols with such goals. 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 further reduce the overheads of pre- and post-processing.

Three protocols stand out in terms of pre-processing overhead — shift left with private offset, and divisions with either public or private divisor. This shows that they make use of many trusted bits, possibly from large rings. We hope that it is possible to verify them with smaller overheads. Indeed, both division protocols may be verified through multiplication, making use of the active privacy of Sharemind's protocols [43]. The current protocol for shifting left is probably sub-optimal and still contains a number of local conversions between elements of $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$.

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 [23] 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 [19, 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 $\mathbb{Z}_p$.

C Full Security Proofs

In this section, we formalize the subprotocols of Sec. 4, and also the main protocol implementing $\mathcal{F}_{vmpc}$ that we defined in Sec. 3. We then prove Theorem 2, which is the generalization of Theorem. 1 of Sec. 3 with more details. Throughout this section, we use $\mathcal{A}$ to denote the real adversary, and $\mathcal{A}\mathsf{s}$ the ideal adversary.

Theorem 2. *Let n be the number of parties. Let $\mathcal{C}$ be the set of covertly corrupted parties, $|\mathcal{C}| < n/2$. Assuming that there is a PKI fixing the public keys of all parties, and a signature scheme with probability of existential forgery δ, there exists a protocol Π_{vmpc} UC-emulating an r-round functionality $\mathcal{F}_{vmpc}$ with correctness error $\varepsilon \leq n^2(3n + r + 6) \cdot \delta + 2^{-\eta}$ for a security parameter η. If the initial protocol of $\mathcal{F}_{vmpc}$ has M_x, M_r, M_c, bits of inputs, randomness, and communication respectively, its circuits have N_b gates requiring bit decompositions, N_r other non-linear gates (treating rotations of length ℓ as ℓ distinct gates), and its largest used ring has cardinality m, then the resulting protocol Π_{vmpc} has at most $11 + r + 6$ rounds, and the communication of different phases has the following upper bounds (let $N_g := N_b + N_r$, $\mathsf{m} := \log m$, and sh_n the number of times the bit width of the value shared among n parties is smaller than the bit width of its one share).*

- *Preprocessing:* $\mathsf{sh}_n \cdot (4n^2\eta\mathsf{m}(N_b\mathsf{m} + N_r) + n^2 M_r + o(n\eta\mathsf{m}(N_b\mathsf{m} + N_r)))$.
- *Execution:* $\mathsf{sh}_n \cdot (n \cdot M_x + M_c + o(rn^2))$.
- *Postprocessing:* $\mathsf{sh}_n \cdot (3n^2(n + 1)N_g\mathsf{m} + o(n^3(N_g + M_c)))$.

If some corrupted party starts deviating from the protocol, the number of rounds may at most double, and the communication may increase at most $2n$ times.

We now give formal definitions of the subprotocols of Sec. 4 and their security proofs.

$\mathcal{F}_{transmit}$ works with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver (if no forwarding is foreseen then $f(id) = r(id)$, and for broadcasts the values of $r(id)$ and $f(id)$ do not matter).
Initialization: On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$ from all (honest) parties, where $\hat{s}$,$\hat{r}$,$\hat{f}$ are mappings s.t $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$, assign $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$.
Secure transmit: On input $(\mathsf{transmit}, id, m)$ from $P_{s(id)}$ and $(\mathsf{transmit}, id)$ from all (honest) parties:

1. Store $(id, m, r(id))$, mark it as undelivered, and output $(id, |m|)$ to $\mathcal{A}_S$.
2. For $s(id) \in \mathcal{C}$, m is chosen by $\mathcal{A}\mathsf{s}$. If the input of $P_{s(id)}$ is invalid (or there is no input), and $r(id) \notin \mathcal{C}$, then output $(\mathsf{corrupt}, s(id))$ to all parties.

Secure broadcast: On input $(\mathsf{broadcast}, id, m)$ from $P_{s(id)}$ and $(\mathsf{broadcast}, id)$ from all (honest) parties:

1. Store (id, m, bc), mark it as undelivered, output $(id, |m|)$ to $\mathcal{A}_S$.
2. For $s(id) \in \mathcal{C}$, m is chosen by $\mathcal{A}\mathsf{s}$. If the input of $P_{s(id)}$ is invalid, output $(\mathsf{corrupt}, s(id))$ to all parties.

Synchronous delivery: At the end of each round:

1. For each undelivered (id, m, r) send (id, m) to P_r; mark (id, m, r) as delivered.
2. For each undelivered (id, m, bc), send (id, m) to each party and $\mathcal{A}_S$; mark (id, m, bc) as delivered.

Forward received message: On input $(\mathsf{forward}, id)$ from $P_{r(id)}$ and on input $(\mathsf{forward}, id)$ from all (honest) parties, after (id, m) has been delivered to $P_{r(id)}$:

1. Store $(id, m, f(id))$, mark as undelivered, output $(id, |m|)$ to $\mathcal{A}_S$.
2. For $s(id), r(id) \in \mathcal{C}$, m is chosen by $\mathcal{A}\mathsf{s}$. If the input of $P_{r(id)}$ is invalid, and $f(id) \notin \mathcal{C}$, output $(\mathsf{corrupt}, r(id))$ to all parties.

Reveal received message: On input (reveal, id) from all (honest) parties, such that $P_{f(id)}$ at any point received (id, m), output (id, m) to each party, and also to $\mathcal{A}_S$.
Do not commit corrupt to corrupt: If for some id both $P_{s(id)}$, $P_{r(id)}$ are corrupt, then on input $(\mathsf{forward}, id)$ the adversary can ask $\mathcal{F}_{transmit}$ to output (id, m') to $P_{f(id)}$ for any m'. If additionally $P_{f(id)}$ is corrupt, then on input (reveal, id) the adversary can ask $\mathcal{F}_{transmit}$ to output (id, m') to all honest parties.

Fig. 5: Ideal functionality $\mathcal{F}_{transmit}$

C.1 Ensuring Message Delivery

From the informal construction of Sec. 4.1, we may abstract away the functionality $\mathcal{F}_{transmit}$ that we will use to ensure message delivery. It is given in Fig. 5. In addition to transmitting a message between two parties, it also allows to broadcast messages, and to forward and reveal previously transmitted messages.

The protocol $\Pi_{transmit}$ implementing $\mathcal{F}_{transmit}$ is formalized in Fig. 6. It works on top of signatures.

The definitions of $\mathcal{F}_{transmit}$ and $\Pi_{transmit}$ are almost directly taken from [21], and defining them is not a contribution of this work. We use this functionality as a building block for our protocols.

From the definition of $\Pi_{transmit}$ in Fig. 6, we can count the number of rounds and the communicated bits of different operations.

Observation 1 *Let c be the number of bits in a signature. The round and bit communication complexities of applying different functions of $\Pi_{transmit}$ to an N-bit message are given in Table. 5.*

We note that there should formally be reserved an additional "empty" round for the cheap mode. This would be a certain time span within which the parties are waiting for possible complaints, and that would be silent in the optimistic setting, when no one attempts to cheat. Since the adversary is covert, we may assume that the accusations, if any, can be as well handled in the next round. Hence everywhere in the cheap mode we have one less round than it may seem from the description of $\Pi_{transmit}$. For similar reasons, in the broadcast and the revealing functionalities, we include the round of exchanging $m \neq m'$ messages only into the expensive mode.

In $\Pi_{transmit}$, each party works locally with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver.

Initialization: On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$, where $\mathsf{Dom}(s) = \mathsf{Dom}(r) = \mathsf{Dom}(f)$, each (honest) party, assigns $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$.

Secure transmit:

1. *Cheap mode:* use as far as $P_{s(id)}$ follows the protocol.
   (a) On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to $P_{r(id)}$.
   (b) On input $(\mathsf{transmit}, id)$ the party $P_{r(id)}$ waits for one round and then expects a message (id, m, σ_s) from $P_{s(id)}$, where σ_s is a valid signature from $P_s(id)$ on (id, m). If it receives it, it outputs (id, m) to $P_{r(id)}$. If it does not receive it, it broadcasts a message $(\mathsf{bad}, s(id))$ using secure broadcast, and upon receiving it, each party goes to the expensive mode.
2. *Expensive mode:* use if $P_{r(id)}$ complains about $P_{s(id)}$ not following the protocol.
   (a) On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to each other party.
   (b) Each (honest) party P_i sends (id, m, σ_s) to $P_{r(id)}$. If P_i does not receive (id, m, σ_s), it sends a signature γ_i on $(\mathsf{corrupt}, s(id))$ to all parties.
   (c) On input $(\mathsf{transmit}, id)$, $P_{r(id)}$ expects a message (id, m, σ_s) from each P_i, where σ_s is a valid signature of $P_{s(id)}$ on (id, m). If it arrives from some P_i, then $P_{r(id)}$ outputs (id, m).

Secure broadcast:

1. On input $(\mathsf{broadcast}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s and sends (id, m, σ_s) to each other party.
2. On input $(\mathsf{broadcast}, id)$ each (honest) party P_i waits for one round and then expects a message (id, m, σ_s) from $P_{s(id)}$, where σ_s is a valid signature from $P_s(id)$ on (id, m). If no message arrives or the signature is invalid, it sends a signature γ_i on $(\mathsf{corrupt}, s(id))$ to each other party. Otherwise, it sends the message (m, id, σ_s) to each other party.
3. If any party receives (id, m, σ_s) and (id, m', σ'_s) for $m \neq m'$, it sends $(id, m, m', \sigma_s, \sigma'_s)$ to each other party.
4. If indeed $m \neq m'$ and the signatures are valid, the honest party P_i receiving them outputs $(\mathsf{corrupt}, s(id))$ to P_i. But if P_i receives only messages (id, m, σ_s) with valid σ_s and no message (id, m', σ'_s) with $m \neq m'$ and valid σ'_s, then it outputs (id, m) to P_i.

Forward received message:

1. On input $(\mathsf{forward}, id)$ the party $P_{r(id)}$ that at some point received (id, m, σ_s) signs (id, m, σ_s) to obtain signature σ_r and sends $(id, m, \sigma_s, \sigma_r)$ to $P_{f(id)}$.
2. On input $(\mathsf{forward}, id)$ the party $P_{f(id)}$ waits for one round and then expects a message $(id, m, \sigma_s, \sigma_r)$ from $P_{r(id)}$, where σ_s $[\sigma_r]$ is a valid signature from $P_{s(id)}$ $[P_{r(id)}]$ on (id, m). If it receives it, it outputs (id, m) to $P_{f(id)}$.

Reveal received message:

1. *Cheap mode:*
   (a) On input (reveal, id), the party $P_{s(id)}$ which at any point sent the message (id, m, σ_s), sends $(\mathsf{reveal}, id, m, \sigma_s)$ to each other party.
   (b) Each (honest) party in turn sends the message to each other party. Several different messages and signatures corresponding to the same id (including the messages $(id, m', \sigma'_s, \sigma'_r)$ that $P_{f(id)}$ may have received before) are handled by an honest party in the same way as by broadcasting, but instead of stopping the protocol, the parties go to the expensive mode. If only a single (id, m, σ_s) is received, an honest party P_i outputs (id, m) to P_i.
2. *Expensive mode:*
   (a) On input (reveal, id), the party $P_{f(id)}$ which at any point received the message $(id, m, \sigma_s, \sigma_r)$, signs $(id, m, \sigma_s, \sigma_r)$ to obtain σ_f and sends $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r, \sigma_f)$ to each other party.
   (b) Each (honest) party in turn sends the message to each other party. Several different messages and signatures corresponding to the same id are handled by an honest party in the same way as by broadcasting. If only a single $(id, m, \sigma_s, \sigma_r, \sigma_f)$ is received, an honest party P_i outputs (id, m) to P_i.

Fig. 6: Real Protocol $\Pi_{transmit}$

Secure transmit:

1. *Cheap mode:*
   (a) On input $(\mathsf{transmit}, id, m)$ for $s(id) \in \mathcal{C}$, $\mathcal{S}$ receives m^* and σ_s^* from $\mathcal{A}$. In its local copy of $\Pi_{transmit}$, it plays sending (id, m^*, σ_s^*) to $P_{r(id)}$. For $s(id), r(id) \notin \mathcal{C}$, $\mathcal{S}$ gets the message length $|m|$ from $\mathcal{F}_{transmit}$. This is needed to model the view of $\mathcal{A}$ on messages moving through secure point-to-point channels between the honest parties. For simplicity, we will ignore $|m|$ in further simulations.
   (b) On input $(\mathsf{transmit}, id)$ for $s(id) \in \mathcal{C}$ and $r(id) \notin \mathcal{C}$, if $\mathcal{A}$ decides not to send a valid message from $s(id)$, then $\mathcal{S}$ plays the broadcast of (bad, id) of $P_{r(id)}$, and goes to the expensive mode. For $s(id) \notin \mathcal{C}$, $r(id) \in \mathcal{C}$, $\mathcal{S}$ receives a message (id, m) from $\mathcal{F}_{transmit}$. It creates a signature σ_m on m and plays delivery of (id, m, σ_s) to $P_{r(id)}$.
2. *Expensive mode:*
   (a) On input $(\mathsf{transmit}, id, m)$, $\mathcal{S}$ signs (id, m) to obtain signature σ_s. $\mathcal{S}$ knows m since if the expensive mode is entered, then either the sender or the receiver is corrupt. $\mathcal{S}$ models sending (id, m, σ_s) to each other party.
   (b) $\mathcal{S}$ models the honest behaviour of $i \notin \mathcal{C}$. For $i \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to P_i.
   (c) $\mathcal{S}$ models the honest behaviour of $r(id) \notin \mathcal{C}$. For $r(id) \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to $P_{r(id)}$. If $(\mathsf{corrupt}, k)$ should be output by any party, then $\mathcal{S}$ sends (stop, k) to $\mathcal{F}_{transmit}$.

Secure broadcast:

1. On input $(\mathsf{broadcast}, id, m)$ for $s(id) \notin \mathcal{C}$, $\mathcal{S}$ receives (id, m) from $\mathcal{F}_{transmit}$ and generates a signature σ_m on m. For $s(id) \in \mathcal{C}$, $\mathcal{S}$ receives m^* and σ_s^* from $\mathcal{A}$. It models sending (id, m, σ_s) (or (id, m^*, σ_s^*)) to each other party.

2-4 For the next broadcast rounds, $\mathcal{S}$ models the honest behaviour of all $i \notin \mathcal{C}$. For $i \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to P_i. If $(\mathsf{corrupt}, k)$ should be output by any party, then $\mathcal{S}$ sends (stop, k) to $\mathcal{F}_{transmit}$.

Forward received message:

1. On input $(\mathsf{forward}, id)$ for $r(id) \notin \mathcal{C}$ and $f(id) \in \mathcal{C}$, $\mathcal{S}$ receives (id, m) from $\mathcal{F}_{transmit}$, generates the signatures σ_s, σ_r on m, and models sending $(id, m, \sigma_s, \sigma_r)$ to $P_{f(id)}$. For $r(id) \in \mathcal{C}$ and $f(id), s(id) \notin \mathcal{C}$, $\mathcal{S}$ should ensure delivery of m that was sent by $s(id)$ on some point. $\mathcal{A}$ may choose some $m^* \neq m$ to be forwarded, and the signatures σ_s^*, σ_r^* on m^*.
2. $\mathcal{S}$ models the behaviour of $P_{f(id)}$ as it did on input $(\mathsf{transmit}, id)$, going to the expensive mode if necessary.

Reveal received message:

1. *Cheap mode:* Simulated similarly to the broadcast. The only difference is that, as soon as a message $(\mathsf{corrupt}, k)$ is received by a party P_i, it goes to the expensive mode instead of stopping the protocol and outputting $(\mathsf{corrupt}, k)$ to $\mathcal{Z}$.
2. *Expensive mode:*
   (a) On input (reveal, id) for $f(id) \notin \mathcal{C}$, $\mathcal{S}$ gets (id, m) from $\mathcal{F}_{transmit}$ and creates the signatures σ_s of $P_{s(id)}$, σ_r of $P_{r(id)}$, and σ_f of $P_{f(id)}$ on m. For $f(id) \in \mathcal{C}$, $\mathcal{S}$ receives m^*, σ_s^*,σ_r^*, σ_f^* from $\mathcal{A}$. $\mathcal{S}$ models sending $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r, \sigma_f)$ (or $(id, m^*, \sigma_s^*, \sigma_s^*, \sigma_f^*)$) to each other party.
   (b) $\mathcal{S}$ models the honest behaviour of all $i \notin \mathcal{C}$. For $i \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to P_i. If $(\mathsf{corrupt}, k)$ should be output by any party, then $\mathcal{S}$ sends (stop, k) to $\mathcal{F}_{transmit}$.

Fig. 7: Simulator $\mathcal{S}_{transmit}$

Table 5. Costs of different functionalities of $\Pi_{transmit}$ applied to N-bit messages, using c-bit signatures

Cheap mode (as far as all parties follow the protocol)		
functionality	rounds	communicated bits
transmit	1	$N + c$
broadcast	2	$n(n-1) \cdot (N + c)$
forward	1	$N + 2c$
reveal	2	$n(n-1) \cdot (N + c)$
Expensive mode (if some party deviates from the protocol)		
functionality	rounds	communicated bits
transmit	2	$2(n-1) \cdot (N + c)$
broadcast	3	$(n(n-1) + 2(n-1)^2) \cdot (N + c)$
forward	2	$2(n-1) \cdot (N + 2c)$
reveal	3	$(n(n-1) + 2(n-1)^2) \cdot (N + 3c)$

Lemma 3. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$ and existence of signature scheme with probability of existential forgery δ, the protocol $\Pi_{transmit}$ UC-realizes $\mathcal{F}_{transmit}$ with correctness error $\varepsilon < N \cdot \delta$ and simulation error 0, where N is the total number of sent messages.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{transmit}$ described in Fig. 7. The simulator runs a local copy of $\Pi_{transmit}$. It also generates signing and verification keys for all the n parties, using a pre-agreed signature scheme.

Simulatability In $\Pi_{transmit}$, the real adversary $\mathcal{A}$ needs to get all the messages received by the corrupted parties. Any message m that is sent to a corrupted party is delivered by $\mathcal{F}_{transmit}$ to $\mathcal{S}$. It is sufficient to know the message length $|m|$ to simulate secure channels between honest parties. For the additional rounds in the expensive mode, $\mathcal{S}$ needs the message m to simulate resolving the conflict (i.e let all parties assist in delivery of m). In this case, the value m is known to $\mathcal{S}$ since the expensive mode is entered if either $s(id) \in \mathcal{C}$ or $r(id) \in \mathcal{C}$. In the first case, m is chosen by $\mathcal{A}$. In the latter case, a message (id, m) comes from $\mathcal{F}_{transmit}$. In addition, $\mathcal{S}$ needs to generate the signatures of honest parties on messages m that is receives from $\mathcal{F}_{transmit}$, which is possible since $\mathcal{S}$ has instantiated the signature scheme itself. Hence everything is perfectly simulatable.

Correctness We discuss the correctness of different modes.

– *Transmission (cheap):* As far as all the parties provide valid signatures, the messages in simulated $\Pi_{transmit}$ are delivered in the same way as in $\mathcal{F}_{transmit}$.
– *Broadcast:* We need to ensure that, either each honest party gets the same message m, or all of them output $(\mathsf{corrupt}, k)$ for the same index $k \in \mathcal{C}$. Suppose that $s(id)$ has sent a message $(id, m_i, \sigma^i_{s(id)})$ to the party P_i, for all $i \in [n]$. If P_i does not receive a valid message, it sends a signature γ_i on $(\mathsf{corrupt}, s(id))$ to each other party. Otherwise, it sends $(id, m_i, \sigma^i_{s(id)}, \sigma^i_i)$ to each other party. If any party receives (id, m, σ_s) and (id, m', σ'_s) for $m \neq m'$, it sends $(id, m, m', \sigma_s, \sigma'_s)$ to each other party, proving that $P_{s(id)}$ misbehaved. This situation is possible only if $P_{s(id)}$ has itself generated the contradictory signatures σ_s and σ'_s. Since the signature includes not only the message, but also the current protocol session and the message identifier id, there is no way for $\mathcal{A}$ to take signatures of some previous rounds or sessions. By properties of the signature scheme, $\mathcal{A}$ may succeed in generating σ_s and σ'_s for $m' \neq m$ with probability at most δ. Hence if $m \neq m'$, then $s(id) \notin \mathcal{C}$ will be accused only with probability at most δ, and for $s(id) \in \mathcal{C}$ a message (stop, k) will be sent by $\mathcal{S}$ to $\mathcal{F}_{transmit}$. If no $(id, m, m', \sigma_s, \sigma'_s)$ has been sent for $m \neq m'$, then all honest parties should have obtained the same message (id, m, id).
– *Transmission (expensive):* If something goes wrong, a message (bad, id) will be broadcast to each party. We have just proven that either all honest parties receive (bad, id), or they output $(\mathsf{corrupt}, k)$ if the broadcast fails due to P_k. After that, each party P_i forwards (id, m_i, σ^i_s) that it received to $P_{r(id)}$, sending a $(\mathsf{corrupt}, k)$ if the signature is invalid (similarly to the broadcast). Differently from broadcast, if P_i gets two properly signed, but different messages $m \neq m'$, it does not distribute $(id, m, m', \sigma_s, \sigma'_s)$ to prove that $P_{s(id)}$ is malicious, but just proceeds with either m or m'. This is allowed since in $\mathcal{F}_{transmit}$

the message m for $s(id) \in \mathcal{C}$ is chosen by $\mathcal{A}$s anyway. At this point, the correctness error of the expensive mode is 0.

– *Forwarding:* A party $P_{r(id)}$ that already holds a signature σ_s on m creates one more signature σ_r on m. sending the message to $P_{f(id)}$. If both $s(id), r(id) \in \mathcal{C}$, then they may choose a new message m^* and create arbitrary signatures on it. This is allowed by $\mathcal{F}_{transmit}$, since it does not commit corrupt to corrupt. If $s(id) \notin \mathcal{C}$, then $r(id)$ may generate a signature σ_s^* on some other message m^* with probability at most δ. If $r(id) \notin \mathcal{C}$, then we would not reach forwarding unless σ_s would be a valid message on m. Hence $P_{f(id)}$ gets valid signatures on m only if it is the same m that was transmitted by $P_{s(id)}$ to $P_{r(id)}$, or otherwise the expensive mode is run for forwarding m.
– *Revealing messages (cheap):* in addition to common broadcast messages, $\mathcal{S}$ may need to simulate the complaint of $P_{f(id)}$ that has received $(id, m', \sigma'_s, \sigma'_r)$ at some point before. If $f(id) \notin \mathcal{C}$, then it should be $s(id) \in \mathcal{C}$ since there is a complaint, and $\mathcal{S}$ creates the signatures σ'_s, σ'_r itself, so the complaint will be accepted. If $f(id) \in \mathcal{C}$, then $\mathcal{A}$ chooses $(id, m', \sigma'_s, \sigma'_r)$ for the complaint. Unless both $s(id), r(id) \in \mathcal{C}$, it may come up with valid signatures σ'_s, σ'_r with probability at most δ. Otherwise, an unreasonable complaint will not be accepted. If all $s(id), r(id)mf(id) \in \mathcal{C}$, then $\mathcal{F}_{transmit}$ allows to reveal $m \neq m'$ since it does not commit corrupt to corrupt.
– *Revealing messages (expensive):* The party $P_{f(id)}$ that holds m and the signatures σ_s, σ_r now broadcasts (id, σ_s, σ_r) to each other party. The broadcast itself involves one more signature σ_f of $P_{f(id)}$. The message m is accepted by an honest party iff all the three signatures σ_s, σ_r, σ_f correspond to m, and there is no $m' \neq m$ that is also provided with valid signatures. If $s(id), r(id), f(id) \in \mathcal{C}$, then $\mathcal{F}_{transmit}$ allows to reveal any value (do not commit corrupt to corrupt). If at least one of them is honest, then its signature can be falsified with probability at most δ.

As a summary, for each message identifier id, if $\mathcal{A}$ wants to force $m' \neq m$ to be delivered for $s(id) \notin \mathcal{C}$ [or $r(id) \notin \mathcal{C}$ in the case of forwarding], it should falsify at least the signature of $P_{s(id)}$ [$P_{r(id)}$] on m, which happens with probability at most δ. Alternatively, if $\mathcal{A}$ just wants to cause the honest parties to blame an innocent $P_{s(id)}$, then it should generate another message m' s.t $m \neq m'$, and $\sigma_{m'}$ is a valid signature of $P_{s(id)}$ on m', which also happens with probability at most δ. If the total number of transmitted messages is N, the probability of cheating in at least one of them is at most $N \cdot \delta$. □

Parallelization. If several messages need to be transmitted to the same party in the same round, it is enough to provide just one signature for all of them. The only problem is that, only some of these messages may need to be forwarded or revealed afterwards, and it should be possible to verify if the signature corresponds to that particular message. We note that the signature covering all the messages of one round can be efficiently constructed by computing a Merkle hash tree of the single signatures of all these messages [47]. If the signature should be verified for only one message, it is necessary to reveal the authentication path of that message, which is just taking one node from each level of the tree, and also the one-time public/private key pair for that particular message. In this way, instead of sending n signatures for n messages, it suffices to send just $\lceil \log n \rceil + 3$ signatures.

C.2 Sharing Based Commitments

From the informal construction of Sec. 4.1, we may abstract away the functionality $\mathcal{F}_{share}$ that we will use to generate shared commitments. It is given in Fig. 8.

The protocol Π_{share} implementing $\mathcal{F}_{share}$ works on top of the message transmission functionality $\mathcal{F}_{transmit}$ defined in Sec. C.1. It ensures that all shares will be delivered to all parties, and that the parties may later prove which shares they received. The protocol Π_{share} is given in Fig. 9 and Fig. 10.

From the definition of Π_{share}, we count the number of $\mathcal{F}_{transmit}$ operations being called for different functions. This allows us to estimate the round and the communication complexity based on the implementation of $\mathcal{F}_{transmit}$.

Observation 2 *The number of $\mathcal{F}_{transmit}$ operations for applying different functions of Π_{share} to an N-bit input is given in Table 6, where tr_M, bc_M, fwd_M, rev_M denote the calls of* transmit*,* broadcast*,* forward*,* reveal

$\mathcal{F}_{share}$ works with unique identifiers id, encoding the ring size $m(id)$ in which the value is shared, and the parties $p(id)$ and $p'(id)$ that get the shared value (if there is only one such party, then $p(id) = p'(id)$). It uses a linear (n,t)-threshold sharing scheme with $t = n/2 + 1$, and fixes a set $\mathcal{H} \subseteq [n] \setminus \mathcal{C}$ of t honest parties. The shared vectors are stored in an array $comm$.

Initialization: On input $(\mathsf{init}, \hat{m}, \hat{p}, \hat{p'})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p}) = \mathsf{Dom}(\hat{p'})$, assign the functions $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$. Send the fixed set $\mathcal{H}$ of the selected t honest parties to $\mathcal{A}_S$.

Share: On input $(\mathsf{share}, (x^k)_{k\in[n]}, id)$ from $P_{p(id)}$ and (share, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Write $comm[id] \leftarrow x$ where $x = \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$. If $p(id) \in \mathcal{C}$, then x^k are chosen by $\mathcal{A}_S$.
2. Output $(x^k)_{k\in[n]}$ to $P_{p(id)}$. For $p(id) \neq k$, output x^k to P_k. For all $k \in \mathcal{C}$, send x^k also to $\mathcal{A}_S$.

Mutual Share: On input $(\mathsf{mshare}, (x^k)_{k\in[n]}, id)$ from $P_{p(id)}$ and (mshare, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Repeat the points (1)-(2) of the sharing functionality.
2. Output $(x^k)_{k\in[n]}$ also to $P_{p'(id)}$.

Reshare: On input $(\mathsf{reshare}, (x^k)_{k\in[n]}, id)$ from $P_{p(id)}$ and $(\mathsf{reshare}, x^{*k}, id)$ from each (honest) party P_k, if $comm[id]$ exists, then do nothing. Otherwise:

– If $p(id) \notin \mathcal{C}$, write $comm[id] \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$.
– If $p(id) \in \mathcal{C}$, write $comm[id] \leftarrow \mathsf{declassify}(x^{*k})_{k\in\mathcal{H}}$.

Compute Linear Combination: On input $(\mathsf{lc}, \boldsymbol{c}, \boldsymbol{id}, id')$ from all (honest) parties, where $p'(id_i)$ are the same for all $i \in \{1,\ldots,|\boldsymbol{id}|\}$, for $p' \leftarrow p'(id)$, $m \leftarrow \min(\{m(id_i) \mid i \in \{1,\ldots,|\boldsymbol{id}|\}\})$:

1. Compute $y \leftarrow \sum_{i=1}^{|\boldsymbol{id}|} c_i \cdot comm[id_i]$.
2. Write $comm[id'] \leftarrow y$.
3. Assign $m(id') \leftarrow m$, $p(id') = p'(id') \leftarrow p'$.

Compute Truncation: On input $(\mathsf{trunc}, m', id, id')$ from all (honest) parties, where $m(id) = 2^m$ for some $m \geq m' \in \mathbb{N}$:

1. Compute $y \leftarrow comm[id] \bmod 2^{m'}$.
2. Write $comm[id'] \leftarrow y$.
3. Assign $m(id') \leftarrow m'$, $p(id') = p'(id') \leftarrow p'(id)$.

Weak Open: On input $(\mathsf{weak\_open}, id)$ from all (honest) parties, ask from $\mathcal{A}_S$ whether $comm[id]$ or $\perp$ should be output to each party. If it is $comm[id]$, output it also to $\mathcal{A}_S$.

Open: On input (open, id) from all (honest) parties, output $comm[id]$ to each party and also to $\mathcal{A}_S$.

Stopping: On input (stop, k) from $\mathcal{A}_S$ output $(\mathsf{corrupt}, k)$ to all parties. For all id, define $p'(id) \leftarrow p(id)$ for $p'(id) = k$, and $p(id) \leftarrow p'(id)$ for $p(id) = k$. Do not accept any inputs s.t $p(id) = p'(id) = k$ anymore.

Fig. 8: Ideal functionality $\mathcal{F}_{share}$

In Π_{share}, each party works locally with unique identifiers id, encoding the ring size $m(id)$ in which the value is shared, and the parties $p(id)$ and $p'(id)$ that know the shared value. The parties use a linear (n, t)-threshold sharing scheme with $t = n/2 + 1$. Each party P_k stores its own local copy of an array $comm^k$ into which it writes its shares. It also stores an array $comm$ into which it writes the entire shared values known to it. For the new indices id that will store the new values computed from the committed values, store a term $deriv[id]$ (represented by a tree whose leaves are commitments, and the inner nodes are operations applied to them) to remember in which way they were computed. For the committed values, $deriv[id] = id$.
Initialization: On input $(\mathsf{init}, \hat{m}, \hat{p}, \hat{p'})$ where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p}) = \mathsf{Dom}(\hat{p'})$, each (honest) party, assigns $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$. It defines mappings s, r, and f, such that $s(id^k_{k'}) \leftarrow p(id)$, $r(id^k_{k'}) \leftarrow k$, and $f(id^k_{k'}) \leftarrow k'$, for all $id \in \mathsf{Dom}(p)$, $k, k' \in [n]$. In addition, it defines the senders $s(id^{\mathsf{bc}}_k) \leftarrow k$ for the broadcasts (used for share opening). It sends (init, s, r, f) to $\mathcal{F}_{transmit}$.
Stopping: At any time when $(\mathsf{corrupt}, k)$ comes from $\mathcal{F}_{transmit}$, each (honest) party outputs $(\mathsf{corrupt}, k)$ to $\mathcal{Z}$. It treats P_k as if it has left the protocol, and defines $p'(id) \leftarrow p(id)$ for $p'(id) = k$, and $p(id) \leftarrow p'(id)$ for $p(id) = k$.

Fig. 9: Real Protocol Π_{share} (initialization and stopping)

Table 6. Calls of $\mathcal{F}_{transmit}$ for different functionalities of Π_{share} with N-bit values

functionality	called $\mathcal{F}_{transmit}$ functionalities
share	$n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot N}$
mshare	$n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot N} + n \cdot \mathsf{fwd}_{\mathsf{sh}_n \cdot N}$
reshare	–
weak_open	$n \cdot \mathsf{bc}_{\mathsf{sh}_n \cdot N}$
open	$n \cdot \mathsf{rev}_{\mathsf{sh}_n \cdot N}$
lc, trunc	–

respectively on an M-bit message, and sh_n the number of times the bit width the value shared among n parties is smaller than the bit width of its one share. We note that, at least with the linear (n, t)-threshold schemes used in this work (see Sec. 4.1), the overhead of share sizes is multiplicative w.r.t to the bit length of the shared value, i.e $\mathsf{sh}_n \cdot (M_1 + M_2) = \mathsf{sh}_n \cdot M_1 + \mathsf{sh}_n \cdot M_2$, which means that several values can be shared in parallel without additional overheads to the share size.

Lemma 4. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{share} UC-realizes $\mathcal{F}_{share}$ in $\mathcal{F}_{transmit}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{share}$ described in Fig. 11. The simulator runs a local copy of Π_{share}, together with a local copy of $\mathcal{F}_{transmit}$. It receives the subset $\mathcal{H}$ of t honest parties from $\mathcal{F}_{share}$. Throughout the simulation, the shares $comm^k[id]$ of $p(id) \in \mathcal{C}$ held by $k \in \mathcal{H}$ should comprise the value $comm[id]$ held by $\mathcal{F}_{vmpc}$. This ensures that the parties are committed to the values they have initially shared.

Simulatability $\mathcal{S}$ simulates the messages communicated through $\mathcal{F}_{transmit}$. All messages of Π_{share} involving either a corrupted sender or a corrupted receiver are given to $\mathcal{S}$ by $\mathcal{F}_{share}$. The only private values that are generated by $\mathcal{S}$ itself are the shares $(x^k)_{k \in \mathcal{C}}$ of x belonging to some honest party, that should be delivered to $\mathcal{A}$. By assumption, Π_{share} works with a linear (n, t)-threshold sharing scheme with $t = n/2 + 1$. Assuming $|\mathcal{C}| < n/2$, there are at most $t - 1$ shares that $\mathcal{S}$ needs to simulate. By definition of threshold secret sharing, any set of less than t shares looks uniformly distributed. Hence it is sufficient sample $x^k \xleftarrow{\$} \mathbb{Z}_{m(id)}$.

For the messages moving between honest parties, $\mathcal{S}$ only needs to simulate $\mathcal{F}_{transmit}$, which should output the message length to $\mathcal{A}$. The message length can be derived from the ring $m(id)$ in which the values are shared.

Correctness The delivery of transmitted and broadcast messages in ensured by $\mathcal{F}_{transmit}$. If any $(\mathsf{corrupt}, k)$ message comes from $\mathcal{F}_{transmit}$, then $\mathcal{S}$ discards P_k from its local run of Π_{share}. Since $|\mathcal{C}| > n/2$ and $t = n/2+1$, there is still enough shares to continue running Π_{share} with the values shared by the other parties. The assignments $p'(id) \leftarrow p(id)$ for $p'(id) = k$, and $p(id) \leftarrow p'(id)$ for $p(id) = k$ ensure that the mutual shares of P_k

Share: On input $(\mathsf{share}, (x^k)_{k\in[n]}, id)$:

1. $P_{p(id)}$ writes $comm[id] \leftarrow (x^k)_{k\in[n]}$.
2. $P_{p(id)}$ sends $(\mathsf{transmit}, id_k^k, x^k)$ to $\mathcal{F}_{transmit}$, for each $k \in [n]$.
3. Upon receiving (id_k^k, x^k) from $\mathcal{F}_{transmit}$, P_k writes $comm^k[id] \leftarrow x^k$, $deriv[id] \leftarrow id$.

Mutual Share: On input $(\mathsf{mshare}, (x^k)_{k\in[n]}, id)$:

1. $P_{p(id)}$ writes $comm[id] \leftarrow (x^k)_{k\in[n]}$.
2. $P_{p(id)}$ sends $(\mathsf{transmit}, (id_k^{p'(id)}, x^k)$ to $\mathcal{F}_{transmit}$ for all $k \in [n]$.
3. Upon receiving $(id_k^{p'(id)}, x^k)$ from $\mathcal{F}_{transmit}$, $P_{p'(id)}$ checks if x^k are valid consistent shares. If they are not, $P'_{p'(id)}$ handles all the received messages as invalid, going to the expensive mode of $\mathcal{F}_{transmit}$.
4. $P_{p'(id)}$ sends $(\mathsf{forward}, id_k^{p'(id)})$ to $\mathcal{F}_{transmit}$ for each $k \in [n]$.
5. Upon receiving $(id_k^{p'(id)}, x^k)$ from $\mathcal{F}_{transmit}$, P_k writes $comm^k[id] \leftarrow x^k$, $deriv[id] \leftarrow id$.

Reshare:

1. On input $(\mathsf{reshare}, (x^k)_{k\in[n]}, id)$, $P_{p(id)}$ writes $comm[id] \leftarrow (x^k)_{k\in[n]}$.
2. On input $(\mathsf{reshare}, x^k, id)$, P_k writes $comm^k[id] \leftarrow x^k$, $deriv[id] \leftarrow id$.

Compute Linear Combination: On input $(\mathsf{lc}, \boldsymbol{c}, \boldsymbol{id}, id')$, where $p'(id_i)$ are the same for all $i \in \{1, \dots, |\boldsymbol{id}|\}$, for $p' \leftarrow p'(id)$, $m \leftarrow \min(\{m(id_i) \mid i \in \{1, \dots, |\boldsymbol{id}|\}\})$, each (honest) P_k

1. computes $y^k \leftarrow \sum_{i=1}^{|\boldsymbol{id}|} c_i \cdot comm^k[id_i] \bmod m$ ($P_{p(id)}$ computes all $(y^k)_{k\in[n]}$);
2. writes $comm^k[id'] \leftarrow y^k$ ($P_{p(id)}$ also writes $comm[id'] \leftarrow (y^k)_{k\in[n]}$);
3. assigns $m(id') \leftarrow m$, $p(id') = p'(id') \leftarrow p'$, $deriv[id'] \leftarrow \mathsf{lc}(\boldsymbol{c}, \boldsymbol{id})$.

Compute Truncation: On input $(\mathsf{trunc}, m', id, id')$, where $m(id) = 2^m$ for some $m \geq m' \in \mathbb{N}$, each (honest) P_k

1. computes $y^k \leftarrow comm^k[id] \bmod 2^{m'}$ ($P_{p(id)}$ computes all $(y^k)_{k\in[n]}$);
2. writes $comm^k[id'] \leftarrow y^k$ ($P_{p(id)}$ also writes $comm[id'] \leftarrow (y^k)_{k\in[n]}$);
3. assigns $m(id') \leftarrow m$, $p(id') = p'(id') \leftarrow p(id)$, $deriv[id'] \leftarrow \mathsf{trunc}(m', id)$.

Weak Open: On input $(\mathsf{weak\_open}, id)$:

1. P_k takes $x^k \leftarrow comm[id]$ and sends $(\mathsf{broadcast}, id_k^{\mathsf{bc}}, x^k)$ to $\mathcal{F}_{transmit}$ for all $k \in [n]$.
2. Upon receiving all shares $(id_k^{\mathsf{bc}}, x^k)$ from $\mathcal{F}_{transmit}$, each (honest) party reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$ and outputs x to $\mathcal{Z}$. If not all n shares come, or they are inconsistent, output $\perp$ to $\mathcal{Z}$.

Open: On input (open, id'):

1. $P_{p(id)}$ sends $(\mathsf{reveal}, id_k^{k'})$ to $\mathcal{F}_{transmit}$ for all endpoint indices $id_k^{k'}$ of the derivation term $deriv[id']$.
2. Upon receiving all $(id_k^{k'}, x^k)$ from $\mathcal{F}_{transmit}$ for all $k \in [n]$, each (honest) party reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$, computes y by instantiating the leaves $id_k^{k'}$ of the term $deriv[id']$ with x^k, and outputs y to $\mathcal{Z}$.

Fig. 10: Real Protocol Π_{share} (sharing, local operations on shared values, and opening)

Let $comm$ be the local array of $\mathcal{F}_{share}$, and $comm^k$, $k \in [n]$ the local arrays of $\mathcal{S}$ that it stores for each party. Let $\mathcal{H}$ be some set of t honest parties.
Share: On input (share, x, id) for $p(id) \in \mathcal{C}$, $\mathcal{S}$ gets the shares $(x^k)_{k\in[n]}$ from $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ generates the shares $x^k \xleftarrow{\$} \mathbb{Z}_{m(id)}$ for $k \in \mathcal{C}$. $\mathcal{S}$ simulates using $\mathcal{F}_{transmit}$ to distribute the shares $(x^k)_{k\in\mathcal{C}}$.
If no $(\mathsf{corrupt}, k)$ has come from $\mathcal{F}_{transmit}$, then all the shares x^k have been successfully delivered. $\mathcal{F}_{share}$ is now waiting for x from $\mathcal{S}$ for $p(id) \in \mathcal{C}$. It may happen that the shares $(x^k)_{k\in[n]}$ coming from $\mathcal{A}$ are inconsistent. $\mathcal{S}$ defines $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, which is unique since $|\mathcal{H}| = t$. It sends x to $\mathcal{F}_{share}$ that writes $comm[id] \leftarrow x$. $\mathcal{S}$ writes $comm[id] = x$, and $comm^k[id] \leftarrow x^k$ for all $k \in [n]$.
Mutual Share: On input (mshare, x, id):

1. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ gets $(x^k)_{k\in[n]}$ from $\mathcal{A}$.
2. If $p'(id) \in \mathcal{C}$, then $\mathcal{S}$ gets $(x^k)_{k\in[n]}$ from $\mathcal{F}_{share}$.
3. Otherwise, it generates $x^k \xleftarrow{\$} \mathbb{Z}_{m(id)}$ for all $k \in \mathcal{C}$.

$\mathcal{S}$ handles the obtained shares similarly to the (share, x, id) case.
Reshare: On input $(\mathsf{reshare}, x^k, id)$, $\mathcal{S}$ receives x^{*k} for $k \in \mathcal{C}$ from $\mathcal{A}$, and just simulates outputting the same x^{*k} back to parties.
Weak Open: On input $(\mathsf{weak\_open}, id)$, $\mathcal{S}$ gets $(x^k)_{k\in[n]}$ from $\mathcal{F}_{share}$. If $k \in \mathcal{C}$, then x^k is chosen by $\mathcal{A}$. $\mathcal{S}$ simulates $(\mathsf{broadcast}, id_k^{\mathsf{bc}}, x^k)$ using $\mathcal{F}_{transmit}$. It models the honest behaviour of $i \notin \mathcal{C}$, and for $i \in \mathcal{C}$ it acts as $\mathcal{A}$ tells to P_i.
Open: On input (open, id), $\mathcal{S}$ gets $(x^k)_{k\in[n]}$ from $\mathcal{F}_{share}$. If $p(id) \in \mathcal{C}$, then $(x^k)_{k\in[n]}$ is chosen by $\mathcal{A}$. $\mathcal{S}$ simulates all the revealings $(\mathsf{reveal}, id_k^{k'}, (x^k)_{k\in[n]})$ of the leaves of $deriv[id]$ using $\mathcal{F}_{transmit}$. It models the honest behaviour of $i \notin \mathcal{C}$, and for $i \in \mathcal{C}$ it acts as $\mathcal{A}$ tells to P_i.
Stopping: $\mathcal{S}$ models the honest behaviour of all $i \notin \mathcal{C}$. For $i \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to P_i. If $(\mathsf{corrupt}, k)$ should be output to each honest party in Π_{share}, then $\mathcal{S}$ discards P_k from its local run of Π_{share}, and assigns $p'(id) \leftarrow p(id)$ for $p'(id) = k$, and $p(id) \leftarrow p'(id)$ for $p(id) = k$.
Compute Linear Combination and Truncation: $\mathcal{S}$ locally preforms the computation for all $k \in \mathcal{C}$. No outputs are produced.

Fig. 11: The simulator $\mathcal{S}_{share}$

with the remaining parties will not be lost. $\mathcal{S}$ sends (stop, k) to $\mathcal{F}_{share}$, so that both the real and the ideal worlds blame P_k and do not perform any computations on its values anymore.

The shares issued to each party are coming from uniform distribution also in $\mathcal{F}_{share}$. We now need to show that the opened values are the same in both worlds.

– Let $p(id) \notin \mathcal{C}$. During the opening, at least the shares $(x^k)_{k\notin\mathcal{C}}$ come from $\mathcal{F}_{share}$, and there are at least t such shares that comprise $comm[id]$. $\mathcal{A}$ may tamper with the shares x^k for $k \in \mathcal{C}$. Since $comm[id]$ is already fixed by the shares of honest parties, $\mathcal{A}$ may at most make the opening inconsistent. In the weak opening case, the opening may fail, and $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{share}$. In the strong opening case, $\mathcal{F}_{transmit}$ ensures that the opened shares are indeed those chosen by $P_{p(id)}$.
– Let $p(id) \in \mathcal{C}$. We need to show that $\mathsf{declassify}(comm^k[id])_{k\in\mathcal{H}} = comm[id]$ for the initially fixed set of honest parties $\mathcal{H}$ is maintained throughout the computation, where $comm^k[id]$ are the shares of the local copy of Π_{share} of $\mathcal{S}$, and $comm[id]$ is the inner value of $\mathcal{F}_{share}$. We will prove it by induction on the number of operations that have been applied to the shared values.
  - *Base:* The initial values for $comm[id]$ are chosen by $\mathcal{A}$ during executing share and mshare. In both cases, $\mathcal{S}$ sends to $\mathcal{F}_{share}$ the value $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, where $x^k = comm^k[id]$ for $k \in \mathcal{H}$ in the local copy of Π_{share} of $\mathcal{S}$. Hence $\mathsf{declassify}(comm^k[id])_{k\in\mathcal{H}} = comm[id]$.
  - *Step:* The new values $comm[id']$ are created when calling $\mathsf{reshare}$, lc and trunc. If $\mathsf{reshare}$ is called for $p(id) \in \mathcal{C}$, then $\mathcal{F}_{share}$ takes the shares x^{*k} of the honest parties to reconstruct $comm[id]$. Since $\mathsf{declassify}(comm^k[id])_{k\in\mathcal{H}} = comm[id]$ held before by induction hypothesis, it still holds. Since both lc and trunc are linear operations, and we are using linear secret sharing, we have

$$\begin{aligned}\mathsf{declassify}(comm^k[id'])_{k\in\mathcal{H}} &= \mathsf{declassify}(f(comm^k[id]))_{k\in\mathcal{H}} \\ &= f(\mathsf{declassify}(comm^k[id]))_{k\in\mathcal{H}} \ .\end{aligned}$$

Table 7. Number of tuple bits involved in different steps (ring cardinality m, length ℓ)

x	$\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell)$	$\mathsf{nbits}_{\mathsf{open}_1}(x,m,\ell)$	$\mathsf{nbits}_{\mathsf{open}_2}(x,m,\ell)$
bit	$\log m$	1	$\log m$
triple	$3\log m$	$2\log m$	$\log m$
cv	$\log\ell + \ell\log m$	$\log\ell$	$\ell\cdot\log m$
rot	$\log\ell + 3\ell\cdot\log m$	$\log\ell + \ell\cdot\log m$	$2\ell\cdot\log m$

By induction hypothesis, $\mathsf{declassify}(comm^k[id]))_{k\in\mathcal{H}} = comm[id]$, and hence this quantity equals $f(comm[id]) = comm[id']$, so $\mathsf{declassify}(comm^k[id'])_{k\in\mathcal{H}} = comm[id']$.

As the result, $\mathcal{A}$ may tamper with the shares x^k for $k \in \mathcal{C}$, but since $comm[id]$ is already fixed by the shares issued to $\mathcal{H}$, they may at most make the opening inconsistent. In the weak opening case, $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{share}$, and the opening fails. In the strong opening case, $\mathcal{F}_{transmit}$ ensures that only the shares $x^k = comm^k[id]$ that have been indeed received by P_k are opened for $k \notin \mathcal{C}$, and so for $k \in \mathcal{H}$. Finally, either $comm[id]$ or $(\mathsf{corrupt}, p(id))$ is output to each party. □

C.3 Generation of Preprocessed Tuples

Fig. 2 of Sec. 4.2 presents the functionality $\mathcal{F}_{pre}$ that we use to generate and share among the parties a sufficient amount of verified preprocessed multiplication triples and trusted bits. We now give in Fig. 12 an extended version of $\mathcal{F}_{pre}$ that also allows to generate the characteristic vectors (CV) and the rotation tuples that we briefly discussed in Sec. 5.1 (verifiable shuffle is omitted since it works only for the 3-party case, but its implementation and proofs would be very similar to the rotation tuples). The protocol Π_{pre} implementing such an extended $\mathcal{F}_{pre}$ is formalized in Fig. 13. It works on top of the sharing functionality $\mathcal{F}_{share}$ defined in Sec. C.2.

Observation 3 *From the definition of Π_{pre}, we can extract the total number of bits $\mathsf{nbits}_{\mathsf{tuple}}(T)$ of a single tuple of type T, and the total numbers $\mathsf{nbits}_{\mathsf{open}_1}(T)$ and $\mathsf{nbits}_{\mathsf{open}_2}(T)$ of tuple bits opened in the pairwise check, where $\mathsf{nbits}_{\mathsf{open}_1}(T)$ bits are opened before the last $\mathsf{nbits}_{\mathsf{open}_2}(T)$ bits. For the rotation tuples, the characteristic vector part and the rotation part can be done in parallel, so two sequential broadcasts are sufficient for each type of tuples. These values are represented by Table 7.*

Lemma 5 (cost of preprocessed tuple generation of Π_{pre}). *Let the $\mathcal{F}_{share}$ used by Π_{pre} be realized by the protocol Π_{share}. Let c be the number of bits in the randomness seed used by the parties, and let the hash function H outputs also be c-bit values. Given the parameters μ and κ, the number of $\mathcal{F}_{transmit}$ operations for generating N tuples of type T using Π_{pre} can be expressed as the quantity* $\mathsf{pre}(N,T) = t\cdot\mathsf{bc}_c + t\cdot\mathsf{bc}_c + n\cdot\mathsf{tr}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{tuple}}T)} + \mathsf{bc}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{tuple}}T)} + \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{open}_1}T)} + \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{open}_2}T)}$.

Proof. By Obs. 2, the cost of sharing of an M-bit value by $\mathcal{F}_{share}$ is $n\cdot\mathsf{tr}_{\mathsf{sh}_n\cdot M}$, and the cost of weak opening an M-bit value by $\mathcal{F}_{share}$ is $\mathsf{bc}_{\mathsf{sh}_n\cdot M}$. The lc operations do not involve any communication. We interpret the terms in the sum defining $\mathsf{pre}(N,T)$. It covers all the communication for generating all the N triples of type t.

- $t\cdot\mathsf{bc}_c + t\cdot\mathsf{bc}_c$ is the cost of agreeing on a common randomness seed.
- $n\cdot\mathsf{tr}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{tuple}}T)}$ is the cost of sharing the initial unverified tuples among the n parties in parallel.
- $\mathsf{bc}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{tuple}}T)}$ is the cost of cut-and-choose opening. All the κ tuples are opened in parallel.
- $\mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{open}_1}T)}$ and $\mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nbits}_{\mathsf{open}_2}T)}$ are the costs of the pairwise verifications of all the $(\mu-1)$ pairs, which counts the total cost of the two sequential weak openings of this step. □

Lemma 6. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, if $\mu > 1 + \eta/\log N$ and $\kappa > \max\{(N^{1/\mu}+1)\eta, N^{1/\mu}+\mu-1\}$, where N is the total number of generated tuples, the protocol Π_{pre} UC-realizes $\mathcal{F}_{pre}$ in $\mathcal{F}_{share}$-$\mathcal{F}_{transmit}$ hybrid model with correctness error $\varepsilon < 2^{\eta}$, and simulation error 0.*

$\mathcal{F}_{pre}$ works with unique identifiers id, encoding a ring size $m(id)$ in which the tuples are shared, the vector length $\ell(id)$ (needed for characteristic vectors and rotations), the party $p(id)$ that gets all the shares, and the number $n(id)$ of tuples to be generated. It stores an array *comm* of already generated triple shares.
Initialization: On input $(\mathsf{init}, \tilde{\ell}, \tilde{m}, \tilde{n}, \tilde{p})$ from all (honest) parties, where $\mathsf{Dom}(\tilde{\ell}) = \mathsf{Dom}(\tilde{m}) = \mathsf{Dom}(\tilde{n}) = \mathsf{Dom}(\tilde{p})$, assign the functions $\ell \leftarrow \tilde{\ell}$, $m \leftarrow \tilde{m}$, $n \leftarrow \tilde{n}$, $p \leftarrow \tilde{p}$.
Trusted bits: On input (bit, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate a vector of random bits $\boldsymbol{r} \xleftarrow{\$} \mathbb{Z}_2^{n(id)}$.
2. Compute the shares $(\boldsymbol{r}^k)_{k\in[n]}$ of $\boldsymbol{r}$ over $\mathbb{Z}_{m(id)}^{n(id)}$, assign $comm[id] \leftarrow (\boldsymbol{r}^k)_{k\in[n]}$.
3. Output $(\boldsymbol{r}^k)_{k\in[n]}$ to $P_{p(id)}$. For $k \neq p(id)$, output $\boldsymbol{r}^k$ to P_k. For all $k \in \mathcal{C}$, send $\boldsymbol{r}^k$ also to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send all shares $(\boldsymbol{r}^k)_{k\in[n]}$ to $\mathcal{A}_S$.

Multiplication triples: On input (triple, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate $\boldsymbol{r}_x \xleftarrow{\$} \mathbb{Z}_{m(id)}^{n(id)}$, $\boldsymbol{r}_y \xleftarrow{\$} \mathbb{Z}_{m(id)}^{n(id)}$, and compute $\boldsymbol{r}_{xy} = \boldsymbol{r}_x \cdot \boldsymbol{r}_y$.
2. Compute the shares $(\boldsymbol{r}_x^k, \boldsymbol{r}_y^k, \boldsymbol{r}_{xy}^k)_{k\in[n]}$ of $(\boldsymbol{r}_x, \boldsymbol{r}_y, \boldsymbol{r}_{xy})$ over $\mathbb{Z}_{m(id)}^{n(id)}$, assign $comm[id] \leftarrow (\boldsymbol{r}_x, \boldsymbol{r}_y, \boldsymbol{r}_{xy})_{k\in[n]}$.
3. Distribute the shares as in the point (3) of trusted bits.

Characteristic vector (CV) pairs: On input (cv, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate a vector of pairs $(r, \boldsymbol{s}) \in (\mathbb{Z}_{\ell(id)} \times \mathbb{Z}_2^{\ell(id)})^{n(id)}$, s.t $r \xleftarrow{\$} \mathbb{Z}_{\ell(id)}$, and for all $i \in [\ell(id)]$, $s_i = 1$ iff $i = r$.
2. Share each pair $(r, \boldsymbol{s})$ to $(r^k, \boldsymbol{s}^k)_{k\in[n]}$ over $\mathbb{Z}_{\ell(id)} \times \mathbb{Z}_{m(id)}^{\ell(id)}$, assign $comm[id] \leftarrow (r, \boldsymbol{s})$ to $(r^k, \boldsymbol{s}^k)_{k\in[n]}$.
3. Distribute the shares as in the point (3) of trusted bits.

Rotation tuples: On input $(\mathsf{rot}, \boldsymbol{id})$ from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate a CV pair $(r, \boldsymbol{s})$ over $\mathbb{Z}_{\ell(id)} \times \mathbb{Z}_2^{\ell(id)}$. Generate $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{\ell(id)}$. Compute $\boldsymbol{b} \in \mathbb{Z}_{m(id)}^{\ell(id)}$ s.t $b_i = a_{(i+r) \bmod \ell(id)}$ for all $i \in [\ell]$. In total, generate $n(id)$ such quadruples $(r, \boldsymbol{s}, \boldsymbol{a}, \boldsymbol{b})$.
2. Share each tuple to $(r^k, \boldsymbol{s}^k, \boldsymbol{a}^k, \boldsymbol{b}^k)_{k\in[n]}$ over $(\mathbb{Z}_{\ell(id)} \times \mathbb{Z}_{m(id)}^{\ell(id)} \times \mathbb{Z}_{m(id)}^{\ell(id)} \times \mathbb{Z}_{m(id)}^{\ell(id)})$, assign $comm[id] \leftarrow (r^k, \boldsymbol{s}^k, \boldsymbol{a}^k, \boldsymbol{b}^k)_{k\in[n]}$.
3. Distribute the shares as in the point (3) of trusted bits.
4. Distribute the shares as in the point (3) of trusted bits.

Stopping: On input (stop) from $\mathcal{A}_S$, stop the functionality and output $\perp$ to all parties.

Fig. 12: Ideal functionality $\mathcal{F}_{pre}$

In $\mathit{\Pi}_{pre}$, each party works with unique identifiers id, encoding a ring cardinality $m(id)$ in which the tuples are shared, the vector length $\ell(id)$, the party $p(id)$ that gets all the shares, and the number $n(id)$ of tuples to be generated. It uses $\mathcal{F}_{share}$ as a subroutine. The parameters μ and κ depend on the security parameter. The protocol uses a one-way function H.
Initialization: On input $(\mathsf{init}, \tilde{\ell}, \tilde{m}, \tilde{n}, \tilde{p})$ from $\mathcal{Z}$, where $\mathsf{Dom}(\tilde{\ell}) = \mathsf{Dom}(\tilde{m}) = \mathsf{Dom}(\tilde{n}) = \mathsf{Dom}(\tilde{p})$, each (honest) party assigns the functions $\ell \leftarrow \tilde{\ell}$, $m \leftarrow \tilde{m}$, $n \leftarrow \tilde{n}$, $p \leftarrow \tilde{p}$. For each id, it defines a couple of identifiers id_i^k for $k \in [\mu \cdot n(id) + \kappa]$, and $i \in [u]$, where u depends of the type of preprocessed tuple to be generated for this id: 1 for bits, 3 for triples, $\ell(id) + 1$ for CV pairs, $3 \cdot \ell(id) + 1$ for rotation tuples. In this way, if the tuple contains several elements, they are just appended together into one vector. It defines $\hat{m}(id_0^k) \leftarrow \ell(id)$ for the CV pairs and the rotation tuples, and it defines $\hat{m}(id_i^k) \leftarrow m(id)$ for all other indices. For all the indices, it defines $\hat{p}(id_i^k) \leftarrow p(id)$. It sends $(\mathsf{init}, \hat{m}, \hat{p}, \hat{p})$ to $\mathcal{F}_{share}$.
Stopping: If at any time $(\mathsf{corrupt}, k)$ comes from $\mathcal{F}_{share}$ or $\mathcal{F}_{transmit}$, output $\perp$ to $\mathcal{Z}$.

Fig. 13: Real protocol $\mathit{\Pi}_{pre}$ (initialization and stopping)

Trusted bits: On input (bit, id):

1. The party $P_{p(id)}$ generates $(\mu \cdot n(id) + \kappa)$ random bits $b_k \overset{\$}{\leftarrow} \mathbb{Z}_2$. It shares b_k to $(b_k^\ell)_{\ell \in [n]}$ and sends $(\mathsf{share}, (b_k^\ell)_{\ell \in [n]}, id_0^k)$ to $\mathcal{F}_{share}$, which distributes the shares among the parties. Each (honest) party sends (share, id_0^k) to $\mathcal{F}_{share}$. If $(\mathsf{corrupt}, i)$ comes from $\mathcal{F}_{share}$ for some i, output $\perp$.
2. The parties jointly agree on κ random indices denoting the bits that are going to be revealed for each id, defining a vector $\boldsymbol{k}(id)$ of random indices. For such agreement, it suffices that each party (any subset of t parties is actually sufficient) broadcasts a random number r_i (using $\mathcal{F}_{transmit}$), and the randomness seed is taken as $\sum_{i \in [n]} r_i$. In order to ensure that no party adapts its r_j to the other parties' r_j, the hashes of r_j are broadcast first. Each party then constructs all the vectors $\boldsymbol{k}(id)$ locally.
   For $k \in \boldsymbol{k}(id)$, each (honest) party sends $(\mathsf{weak\_open}, id_0^k)$ to $\mathcal{F}_{share}$, getting back a bit b_k. If the opening fails, or $b_k \notin \{0, 1\}$, then output $\perp$.
3. The parties jointly agree on random pairing of $2 \cdot n(id)$ bits. For each pair (k, k'), $P_{p(id)}$ broadcasts a bit indicating whether $b_k = b_{k'}$ or not. If $b_k = b_{k'}$ was indicated, each (honest) party sends $(\mathsf{lc}, [1, -1], [id_0^k, id_0^{k'}], id_0^{k,k'})$ to $\mathcal{F}_{share}$. If $b_k \neq b_{k'}$ was indicated, each (honest) party sends $(\mathsf{lc}, [1, 1], [id_0^k, id_0^{k'}], id_0^{k,k'})$ to $\mathcal{F}_{share}$. Each (honest) party then sends $(\mathsf{weak\_open}, id_0^{k,k'})$ to $\mathcal{F}_{share}$ and checks if the value returned by $\mathcal{F}_{share}$ equals 0 if $b_k = b_{k'}$ was indicated, and 1 if $b_k \neq b_{k'}$ was indicated. If it does not, then output $\perp$. This pairwise verification is repeated $\mu - 1$ times.
4. P_i outputs the remaining $n(id)$ shares b_k^i. $P_{p(id)}$ outputs $(b_k^i)_{i \in [n]}$.

Multiplication triples: On input (triple, id):

1. The party $P_{p(id)}$ generates $(\mu \cdot n(id) + \kappa)$ random ring element pairs $a_k \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}$, $b_k \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}$. It computes $c_k = a_k \cdot b_k$ for $k \in |\mu \cdot n(id) + \kappa|$. Each value is shared similarly to the step (1) of trusted bits, using $\mathcal{F}_{share}$.
2. The parties reveal and check κ random triples similarly to the step (2) of bits.
3. The parties agree on a pairing of $2 \cdot n(id)$ triples, as in the step (3) of trusted bits. For each pair (k, k'), each (honest) party sends $(\mathsf{lc}, [1, -1], [id_0^k, id_0^{k'}], id_0^{k,k'})$, $(\mathsf{lc}, [1, -1], [id_1^k, id_1^{k'}], id_1^{k,k'})$, and then $(\mathsf{weak\_open}, id_0^{k,k'})$, $(\mathsf{weak\_open}, id_1^{k,k'})$ to $\mathcal{F}_{share}$, getting back $\hat{a}$ and $\hat{b}$ respectively. Each (honest) party then sends $(\mathsf{lc}, [\hat{a}, \hat{b}, 1, -1], [id_1^k, id_0^{k'}, id_2^{k'}, id_2^k], id_2^{k,k'})$ and $(\mathsf{weak\_open}, id_2^{k,k'})$ to $\mathcal{F}_{share}$. If $\mathcal{F}_{share}$ outputs a value $z \neq 0$, then output $\perp$. The pairwise verification is repeated $\mu - 1$ times.
4. P_i outputs the remaining $n(id)$ shares (a_k^i, b_k^i, c_k^i). $P_{p(id)}$ outputs $(a_k^i, b_k^i, c_k^i)_{i \in [n]}$.

Fig. 14: Real protocol $\mathit{\Pi}_{pre}$ (basic tuples)

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{pre}$ described in Fig. 16. The simulator runs a local copy of $\mathit{\Pi}_{pre}$, together with local copies of $\mathcal{F}_{transmit}$ and $\mathcal{F}_{share}$.

Simulatability The shares of dishonest parties may be generated uniformly, by reasoning similar to the case of $\mathcal{S}_{share}$, since there are at most $n/2$ shares to be revealed, and we are using linear (n, t)-threshold sharing with $t \geq n/2 + 1$. The simulator needs to generate some non-trivial values during the openings of the cut-and-choose and the pairwise verification.

1. First of all, the randomness r is generated. $\mathcal{S}$ makes its shares r_i dependent on the shares r_i chosen by $\mathcal{A}$. Since the hashes of shares $H(r_i^k)$ should be sent to P_k first, $\mathcal{A}$ cannot adapt its values r_i^k to those of $\mathcal{S}$, unless $\mathcal{A}$ makes them inconsistent with the previously sent hashes (in the latter case $\mathcal{S}$ sends stop to $\mathcal{F}_{pre}$). Hence at the point when $\mathcal{S}$ needs to send its own hashes $H(r_i^k)$ to $\mathcal{A}$, it knows the distribution from which $\mathcal{A}$ generates its r_i, so it just generates the remaining r_i in such a way that their sum equals to the desired value of r such that the finally chosen random tuples will be exactly those chosen by $\mathcal{F}_{pre}$. Since $\mathcal{S}$ initially shuffled the shares, such an r is distributed uniformly.
2. For cut-and-choose of honest provers, $\mathcal{S}$ generates the opened tuples itself from the same distribution as an honest prover would. By choice of the randomness seed r, these tuples are completely new and are not related to the $n(id)$ tuples generated by $\mathcal{F}_{pre}$.
3. For the pairwise verification, $\mathcal{S}$ needs to simulate different values, depending on the tuple type. For the first $\mu - 1$ iterations, $\mathcal{S}$ generates all the tuples for honest parties itself, since they are not included into

Characteristic vector (CV) pairs: On input (cv, id) from all (honest) parties:

1. The party $P_{p(id)}$ generates $(\mu \cdot n(id) + \kappa)$ pairs $(r_k, \boldsymbol{s}_k)$ where $r_k \xleftarrow{\$} \mathbb{Z}_{\ell(id)}$, and $\boldsymbol{s}_k \in \mathbb{Z}_2^{\ell(id)}$ is such that $s_{ki} = 1$ iff $i = r_k$. As in the step (1) of trusted bits, these values are shared and distributed using $\mathcal{F}_{share}$.
2. The parties reveal and check κ random pairs similarly to the step (2) of bits.
3. The parties agree on a pairing of $2 \cdot n(id)$ tuples, as in the step (3) of trusted bits. For each pair (k, k'), each (honest) party sends $(\mathsf{lc}, [1, -1], [id_0^k, id_0^{k'}], id_0^{k,k'})$ and $(\mathsf{weak\_open}, id_0^{k,k'})$ to $\mathcal{F}_{share}$, getting back $\hat{r}$.
   Each (honest) party then sends $(\mathsf{lc}, [1, -1], [id_{i+1}^k, id_{((i+\hat{r}) \bmod \ell)+1}^{k'}], id_{i+1}^{k,k'})$, $(\mathsf{weak\_open}, id_i^{k,k'})$ for $i \in [\ell(id)]$ to $\mathcal{F}_{share}$, getting back values z. If $z \neq 0$ for some z, output $\perp$.
4. P_i outputs the remaining $n(id)$ shares $(r_k^i, \boldsymbol{s}_k^i)$. $P_{p(id)}$ outputs $(r_k^i, \boldsymbol{s}_k^i)_{i \in [n]}$.

Rotation tuples: On input (rot, id) from all (honest) parties:

1. The party $P_{p(id)}$ generates and shares $(\mu \cdot n(id) + \kappa)$ CV pairs $(r_k, \boldsymbol{s}_k)$ over $\mathbb{Z}_{\ell(id)} \times \mathbb{Z}_{m(id)}^{\ell(id)}$. For each such pair, it also generates $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{\ell(id)}$ and computes $\boldsymbol{b} \in \mathbb{Z}_{m(id)}^{\ell(id)}$ s.t $b_i = a_{(i+r) \bmod \ell(id)}$. As in the step (1) of trusted bits, all the entries of the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are shared and distributed using $\mathcal{F}_{share}$.
2. The parties reveal and check κ random tuples similarly to the step (2) of bits.
3. The parties agree on a pairing of $2 \cdot n(id)$ tuples, as in the step (3) of trusted bits. For each pair (k, k'), each (honest) party does the following:
   (a) Similarly to the step (3) of CV pairs, prove that $\boldsymbol{s}_k$ is a characteristic vector of r_k. For each pair (k, k'), each (honest) party sends $(\mathsf{lc}, [1, -1], [id_0^k, id_0^{k'}], id_0^{k,k'})$ and $(\mathsf{weak\_open}, id_0^{k,k'})$ to $\mathcal{F}_{share}$, getting back $\hat{r}$. Each (honest) party sends $(\mathsf{lc}, [1, -1], [id_{i+1}^k, id_{((i+\hat{r}) \bmod \ell)+1}^{k'}], id_{i+1}^{k,k'})$, $(\mathsf{weak\_open}, id_i^{k,k'})$ for $i \in [\ell]$ to $\mathcal{F}_{share}$, getting back values z. If $z \neq 0$ for some z, output $\perp$.
   (b) Send $(\mathsf{lc}, [1, -1], [id_{\ell+i+1}^k, id_{\ell+i+1}^{k'}], id_{\ell+i+1}^{k,k'})$ and $(\mathsf{weak\_open}, id_{\ell+i+1}^{k,k'})$ to $\mathcal{F}_{share}$ for $i \in [\ell]$, getting $\hat{\boldsymbol{a}} = \boldsymbol{a}_k - \boldsymbol{a}_{k'}$.
   (c) Send $(\mathsf{lc}, [1, -1], [id_{2\ell+i+1}^k, id_{((i+\hat{r}) \bmod \ell)+2\ell+1}^{k'}], id_{2\ell+i+1}^{k,k'})$ to $\mathcal{F}_{share}$ for $i \in [\ell]$, expecting the rotation of $\hat{a}$ by r_k (which needs to be checked).
   (d) Rotate $\hat{\boldsymbol{a}}$ by r_k by computing the linear combinations $(\mathsf{lc}, \hat{\boldsymbol{a}}, [id_i^k, \ldots, id_{((\ell+i) \bmod \ell)+1}^k], id_{3\ell+i+1}^{k,k'})$ for all $i \in [\ell(id)]$.
   (e) Send $(\mathsf{lc}, [1, -1], [id_{2\ell+i+1}^{k,k'}, id_{3\ell+i+1}^{k,k'}], id_{4\ell+i+1}^{k,k'})$ and $(\mathsf{weak\_open}, id_{4\ell+i+1}^{k,k'})$ to $\mathcal{F}_{share}$ for all $i \in [\ell(id)]$. If there is at least one opened value $z \neq 0$, output $\perp$.
4. P_i outputs the remaining $n(id)$ shares $(r_k^i, \boldsymbol{s}_k^i, \boldsymbol{a}_k^i, \boldsymbol{b}_k^i)$. $P_{p(id)}$ outputs $(r_k^i, \boldsymbol{s}_k^i, \boldsymbol{a}_k^i, \boldsymbol{b}_k^i)_{i \in [n]}$.

Fig. 15: Real protocol Π_{pre} (extended tuples)

$\mathcal{F}_{pre}$ anyway. The most interesting is the last μ-th iteration. Let k be the tuple that will be finally output and is not known to $\mathcal{S}$, and let k' be the tuple that $\mathcal{S}$ may still choose itself.

(a) *Trusted bits:* First, $\mathcal{S}$ needs to broadcast a bit indicating whether $b_k \neq b_{k'}$. This value is distributed uniformly since $b_{k'}$ has not been used anywhere yet. After that, $\mathcal{S}$ simulates $\mathcal{F}_{share}$ outputting either $b_k - b_{k'}$, or $b_k + b_{k'}$. For an honest prover, that value is always 0 for $b_k - b_{k'}$, and 1 for $b_k + b_{k'}$ since it tells honestly if $b_k \neq b_{k'}$.

(b) *Multiplication triples:* $\mathcal{S}$ broadcasts $\hat{a} = a_k - a_{k'}$ and $\hat{b} = b_k - b_{k'}$ which are uniformly distributed due to the masks $a_{k'}$ and $b_{k'}$. For an honest prover, the value $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + c_{k'} - c_k$ equals 0, since it would generate $c_k = a_k \cdot b_k$ and $c_{k'} = a_{k'} \cdot b_{k'}$.

(c) *CV pairs:* $\mathcal{S}$ broadcasts $\hat{r} = r_k - r_{k'}$, which is uniformly distributed due to the mask $r_{k'}$. After that, the vector $\boldsymbol{s}_{k'}$ rotated by $\hat{r}$ is opened. For an honest prover, rotating $\boldsymbol{s}_{k'}$ back to $r_{k'}$ positions gives a vector $(1, 0, \ldots, 0)$, and rotating it forward by r_k positions should indeed give $\boldsymbol{s}_k$ as the result. Hence all the final opened values are 0.

(d) *Rotation tuples:* The first checks are the same as for the CV tuple, and the masks $r_{k'}$ and $\boldsymbol{s}_{k'}$ are used up at this point. Similarly, $\hat{\boldsymbol{a}} = \boldsymbol{a}_k - \boldsymbol{a}_{k'}$ is uniform due to the mask $\boldsymbol{a}_{k'}$. Finally, the differences between the rotation of $\hat{\boldsymbol{a}}$ by $\boldsymbol{r}_k$ and the differences between $\boldsymbol{b}_k$ and the rotation of $\boldsymbol{b}_{k'}$ by $\hat{r}$ are opened. For an honest prover, rotating $\boldsymbol{b}_{k'}$ by $\hat{r} = r_k - r_{k'}$ equals to rotating $\boldsymbol{a}_{k'}$ by r_k, and since

Tuple generation: On input (bit, id), (triple, id), (cv, id), and, (rot, id), $\mathcal{S}$ behaves according to the following pattern:

1. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ obtains $n(id)$ shares s^k from $\mathcal{F}_{pre}$ for each $k \in \mathcal{C}$. It generates $(\mu - 1)n(id) + \kappa$ more shares for each $k \in \mathcal{C}$, shuffles all the shares, and delivers them to $\mathcal{A}$. If $p(id) \in \mathcal{C}$, then all the $(\mu - 1)n(id) + \kappa$ shares $(s^k)_{k\in[n]}$ are chosen by $\mathcal{A}$. Even if the shares of $\mathcal{A}$ are inconsistent or do not comprise valid tuples, the simulation does not stop yet. $\mathcal{S}$ models distributing s^k to the parties using $\mathcal{F}_{share}$.
2. The parties should jointly agree to reveal some random κ tuples.
   - In order to agree on the same κ, each party broadcasts a random number r_i, and the randomness is taken as $r = \sum_i r_i$. All the communication takes part through $\mathcal{F}_{transmit}$ and hence is easy to simulate. For $i \in \mathcal{C}$, the values r_i are chosen by $\mathcal{A}$. For $i \notin \mathcal{C}$, $\mathcal{S}$ generates $r_i \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}$, so that r in such that the only $n(id)$ tuples that have not been generated by $\mathcal{S}$ are exactly those that remain accepted in the end.
   - Now a vector $\boldsymbol{s}'$ of certain κ tuples needs to be revealed. $\mathcal{S}$ needs to simulates the weak opening of $\mathcal{F}_{share}$, but that requires knowing the values $\boldsymbol{s}'$ to be opened. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ takes the $\boldsymbol{s}'$ chosen by $\mathcal{A}$ before. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ only knows $\boldsymbol{s}'^k$ for $k \in \mathcal{C}$. In this case, $\mathcal{S}$ generates a vector of random valid tuples $\boldsymbol{s}'$, since that is what an honest prover would generate. After $\mathcal{F}_{share}$ returns the published results, $\mathcal{S}$ either accepts or rejects these tuples from the name of honest parties, exactly in the same way as they would do in Π_{pre}. If any tuple should be rejected, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{pre}$.
3. The parties start verifying the tuples pairwise. For this, certain values should be computed and opened using $\mathcal{F}_{share}$, that depend on the tuple type. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ already knows these values, and hence can simulate their opening. If there are any inconsistencies, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{pre}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulates two types of values:
   - The first component are alleged zeroes. For these, $\mathcal{S}$ simulates opening 0.
   - The second component are some additional values needed in verification. For these, $\mathcal{S}$ simulates opening uniformly distributed random values in the corresponding rings.
4. There are now $n(id)$ shares left for each party that are treated as the final output. For $p(id) \in \mathcal{C}$, $\mathcal{F}_{pre}$ has shared $\boldsymbol{s}$ in such a way that $\mathsf{declassify}(\boldsymbol{s}^k)_{k\notin\mathcal{C}}$ are valid tuples. If the cut-and-choose and the pairwise verification have not failed, then the same holds also in Π_{pre} with probability that depends on the security parameters.

Stopping: If at any time $(\mathsf{corrupt}, k)$ should come from $\mathcal{F}_{share}$ or $\mathcal{F}_{transmit}$, output (stop) to $\mathcal{F}_{pre}$.

Fig. 16: The simulator $\mathcal{S}_{pre}$

$\boldsymbol{b}_k$ is $\boldsymbol{a}_k$ rotated by r_k, their difference is indeed $\boldsymbol{a}_k - \boldsymbol{a}_{k'}$ (which is $\hat{\boldsymbol{a}}$) rotated by r_k. Hence all the opened values are 0.

Correctness For $p(id) \notin \mathcal{C}$, the finally left $n(id)$ tuples are exactly those that are generated by $\mathcal{F}_{pre}$. For $p(id) \in \mathcal{C}$, these $n(id)$ shares are all generated by $\mathcal{A}$. We need to show that, if finally $n(id)$ tuples are accepted for $p(id) \in \mathcal{C}$, then they are all valid, except with negligible probability.

First of all, we show that, if the tuple with the index k' is valid, then the pairwise check passes only if the tuple k is also valid. We prove it for different kinds of tuples, one by one.

1. *Trusted bits:* Let $b_{k'} \in \{0, 1\}$. First, the prover decides whether to indicate $b_k = b_{k'}$, or $b_k \neq b_{k'}$.
   - Suppose the prover indicated $b_k = b_{k'}$. In this case, $b_k - b_{k'}$ is output. If indeed $b_k - b_{k'} = 0$, then it should be $b_k = b_{k'} \in \{0, 1\}$.
   - Suppose the prover indicated $b_k \neq b_{k'}$. In this case, $b_k + b_{k'}$ is output. If indeed $b_k + b_{k'} = 1$, then it should be $b_k = 1 - b_{k'} \in \{0, 1\}$.
   - If the prover indicates something else, the protocol aborts. No tuples are accepted.
2. *Multiplication triples:* Let $c_{k'} = a_{k'} \cdot b_{k'}$. The values $\hat{a} = a_k - a_{k'}$ and $\hat{b} = b_k - b_{k'}$ are computed and opened by the parties using $\mathcal{F}_{share}$, so there is no way to cheat with them. Suppose that $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + c_{k'} - c_k = 0$. Since $c_{k'} = a_{k'} \cdot b_{k'}$, we have $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + a_{k'} \cdot b_{k'} - c_k = (a_k - a_{k'}) \cdot b_k + (b_k - b_{k'}) \cdot a_{k'} + a_{k'} \cdot b_{k'} - c_k = a_k \cdot b_k - c_k$. If this value is 0, then $a_k \cdot b_k = c_k$.
3. *CV pairs:* Let $\boldsymbol{s}_{k'}$ be a CV of $r_{k'}$. Since $\hat{r} = r_k - r_{k'}$ is computed by the parties using $\mathcal{F}_{share}$, there is no way to cheat with that value. After that, the parties again use $\mathcal{F}_{share}$ to rotate the vector $\boldsymbol{s}_{k'}$ by $\hat{r}$. Let it be denoted $\boldsymbol{t}$. By correctness of the tuple k', the vector $\boldsymbol{t}$ is indeed a CV of r_k for the shared r_k, and $\boldsymbol{t} - \boldsymbol{s}_k = \boldsymbol{0}$ implies that so is the vector $\boldsymbol{s}_k$.

4. *Rotation tuples:* Let $\boldsymbol{s}_{k'}$ be a CV of $r_{k'}$, and $\boldsymbol{b}_{k'}$ a rotation of $\boldsymbol{a}_{k'}$ by $r_{k'}$. The first CV tuple checks prove that $\boldsymbol{s}_k$ is indeed a CV of r_k. Since $\hat{\boldsymbol{a}} = \boldsymbol{a}_k - \boldsymbol{a}_{k'}$ is computed and opened by the parties using $\mathcal{F}_{share}$, there is no way to cheat with it. Similarly, the rotation of $\hat{\boldsymbol{a}}$ by r_k (denote it $\boldsymbol{c}$), the rotation of $\boldsymbol{b}_{k'}$ by $\hat{r}$ (denote it $\boldsymbol{d}$), and the difference $\boldsymbol{e} = \boldsymbol{b}_k - \boldsymbol{d}$, are all computed using $\mathcal{F}_{share}$. Since $\boldsymbol{b}_{k'}$ is a rotation of $\boldsymbol{a}_{k'}$ by $r_{k'}$, we get that $\boldsymbol{d}$ is $\boldsymbol{a}^{k'}$ rotated by r_k.

   Now suppose that the final checks passed, and $\boldsymbol{e} - \boldsymbol{c} = \boldsymbol{0}$. This means that $\boldsymbol{e}$ is the vector $\hat{\boldsymbol{a}} = \boldsymbol{a}_k - \boldsymbol{a}_{k'}$ rotated by r_k, which we can rewrite as $\boldsymbol{a}_k^r - \boldsymbol{d}$, where $\boldsymbol{a}_k^r$ is $\boldsymbol{a}_k$ rotated by r_k. Since at the same time $\boldsymbol{e} = \boldsymbol{b}_k - \boldsymbol{d}$ (as computed by the parties), we get $\boldsymbol{b}_k = \boldsymbol{a}_k^r$, so $\boldsymbol{b}_k$ is indeed a rotation of $\boldsymbol{a}_k$ by r_k.

We have shown that the only possibility for the prover to cheat is to put two invalid tuples into the same pair. For the $\mu - 1$ pairwise checks, the finally accepted invalid tuple should be paired with some other invalid tuple on each iteration. Now we need to show that, for sufficiently large μ and κ, this happens only with a negligible probability.

Let $p(id) \in \mathcal{C}$. Let $n := n(id)$. If $P_{p(id)}$ wants to have ℓ bad tuples among the final n ones, it has to do the following:

1. put $\mu \cdot \ell$ bad tuples into the initial set of $(\mu \cdot n + \kappa)$ tuples;
2. hope that no bad tuple is among the ones opened during the cut-and-choose step;
3. hope that the $\mu \cdot \ell$ tuples are put together into ℓ groups during the pairwise checking step.

We will now give lower bounds for the values μ and κ, such that the probability of steps (2) and (3) succeeding (from the point of view of a malicious prover) is less than $2^{-\eta}$ for a security parameter η.

Probability of passing cut-and-choose. The κ tuples to be opened can be selected in $\binom{\mu n+\kappa}{\kappa}$ different ways. Assuming that some ℓ of the n tuples are "bad", there are $\binom{\mu(n-\ell)+\kappa}{\kappa}$ ways of choosing a set that contains only "good" tuples. The probability of selecting such a set is

$$P_{\text{c\&c}}(\mu,n,\kappa,\ell) = \frac{\binom{\mu(n-\ell)+\kappa}{\kappa}}{\binom{\mu n+\kappa}{\kappa}} = \frac{(\mu(n-\ell)+\kappa)!}{(\mu n+\kappa)!}\cdot\frac{(\mu n)!}{(\mu(n-\ell))!} = \frac{\mu n\cdots(\mu(n-\ell)+1)}{(\mu n+\kappa)\cdots(\mu(n-\ell)+\kappa+1)} \le \left(\frac{\mu n}{\mu n+\kappa}\right)^{\mu\cdot\ell} \tag{1}$$

In particular, if $\ell \ge cn$ for some $c \in [0,1]$, then, assuming $\kappa \le \frac{\mu n}{2}$,

$$P_{\text{c\&c}}(\mu,n,\kappa,\ell) \le \left(\frac{\mu n}{\mu n+\kappa}\right)^{\mu n c} = \left(\frac{1}{1+\frac{\kappa}{\mu n}}\right)^{\mu n c} = \frac{1}{\left(\left(1+\frac{\kappa}{\mu n}\right)^{\frac{\mu n}{\kappa}}\right)^{c\kappa}} \le \frac{1}{2^{c\kappa}} \tag{2}$$

Probability of passing pairwise checking. For the pairwise checking, we partition the μn tuples into n groups of size μ, so that only one tuple of each group is left after checking. We have $\binom{\mu n}{\mu}$ ways to select the first group, $\binom{\mu n-\mu}{\mu}$ ways to select the second group, $\binom{\mu n-2\mu}{\mu}$ ways to select the third group, etc. If we multiply all these values, we get the number of all possible groupings, where the order of the groups matters. Since the order of the groups is not important, we have to divide the final number by $n!$. The number of groupings of μn tuples into n groups is

$$\mathcal{G}(\mu,n) = \frac{1}{n!}\prod_{i=0}^{n-1}\binom{\mu(n-i)}{\mu} = \frac{1}{n!}\left(\frac{1}{\mu!}\right)\prod_{i=0}^{n-1}\frac{(\mu(n-i))!}{(\mu(n-i-1))!} = \frac{1}{n!}\left(\frac{1}{\mu!}\right)^n\frac{(\mu(n-0))!}{(\mu(n-(n-1)-1))!} = \frac{(\mu n)!}{n!(\mu!)^n} \tag{3}$$

In order to pass the pairwise checking, all the $\mu\ell$ bad tuples should form exactly ℓ groups of size μ, such that no "good" tuple is present in any of these groups. The number of such groupings is $\mathcal{G}(\mu,\ell)\cdot\mathcal{G}(\mu,n-\ell)$, and the probability of this happening is

$$\begin{aligned}P_{\mathrm{pwc}}(\mu,n,\ell) &= \frac{\mathcal{G}(\mu,\ell)\cdot\mathcal{G}(\mu,n-\ell)}{\mathcal{G}(\mu,n)} \\ &= \frac{(\mu\ell)!}{\ell!(\mu!)^{\ell}}\cdot\frac{(\mu(n-\ell))!}{(n-\ell)!(\mu!)^{(n-\ell)}}\cdot\frac{n!(\mu!)^n}{(\mu n)!} \\ &= \frac{n!}{\ell!(n-\ell)!}\cdot\frac{(\mu\ell)!(\mu n-\mu\ell)!}{(\mu n)!} \\ &= \binom{n}{\ell}\Big/\binom{\mu n}{\mu\ell} \qquad (4)\end{aligned}$$

Probability of passing both checks. This probability is the product of (1) and (4):

$$\begin{aligned}P_{\mathrm{pp}}(\mu,n,\kappa,\ell) &= \frac{\binom{\mu(n-\ell)+\kappa}{\kappa}\binom{n}{\ell}}{\binom{\mu n+\kappa}{\kappa}\binom{\mu n}{\mu\ell}} \\ &= \binom{n}{\ell}\cdot\frac{(\mu n-\mu\ell+\kappa)!}{(\mu n-\mu\ell)!\kappa!}\cdot\frac{(\mu n)!\kappa!}{(\mu n+\kappa)!}\cdot\frac{(\mu\ell)!(\mu n-\mu\ell)!}{(\mu n)!} \\ &= \binom{n}{\ell}\Big/\binom{\mu n+\kappa}{\mu\ell} \qquad (5)\end{aligned}$$

Catching a single bad tuple. Suppose that the malicious prover aims to have a single bad tuple among the final n ones, i.e. $\ell = 1$. In this case

$$P_{\mathrm{pp}}(m,n,k,1) = \binom{n}{1}\Big/\binom{\mu n+k}{\mu\cdot 1} \le n\Big/\left(\frac{\mu n+k}{\mu}\right)^{\mu} = n\Big/\left(n+\frac{k}{\mu}\right)^{\mu} \le n^{1-\mu} \quad .$$

In order to have $P_{\mathrm{pp}}(\mu,n,\kappa,1)\le 2^{-\eta}$, it is sufficient to have $n^{1-\mu}\le 2^{-\eta}$ or $\mu \ge 1+\eta/\log n$.

Making a single bad tuple the worst case. We aim to select the parameters μ and κ in such a way, that aiming for a single bad tuple is the best strategy for the malicious prover.

First, let $\ell < cn$ for some $c\in[0,1]$ (fixed below). We consider the ratio $P_{\mathrm{pp}}(\mu,n,\kappa,\ell)/P_{\mathrm{pp}}(\mu,n,\kappa,\ell+1)$ and search for sufficient conditions for it to be larger than 1. Let $a^{\underline{n}} := a(a-1)\cdots(a-n+1)$.

$$\begin{aligned}\frac{P_{\mathrm{pp}}(\mu,n,\kappa,\ell)}{P_{\mathrm{pp}}(\mu,n,\kappa,\ell+1)} &= \frac{\binom{n}{\ell}\binom{\mu n+\kappa}{\mu\ell+\mu}}{\binom{n}{\ell+1}\binom{\mu n+\kappa}{\mu\ell}} \\ &= \frac{(\ell+1)!(n-\ell-1)!}{(n-\ell)!\ell!}\cdot\frac{(\mu n-\mu\ell+\kappa)!(\mu\ell)!}{(\mu n+k-\mu\ell-\mu)!(\mu\ell+\mu)!} \\ &= \frac{(\ell+1)}{(n-\ell)}\cdot\frac{(\mu(n-\ell)+\kappa)^{\underline{\mu}}}{(\mu(\ell+1))^{\underline{\mu}}} \\ &\ge \frac{1}{n}\cdot\left(\frac{\mu(n-\ell-1)+\kappa+1}{\mu\ell+1}\right)^{\mu} \qquad (6)\end{aligned}$$

For (6) to be at least 1, it is sufficient to take

$$\mu(n-\ell-1)+\kappa+1 \ge n^{1/\mu}(\mu\ell+1),$$

meaning that it suffices for κ to be at least

$$\begin{aligned}n^{1/\mu}(\mu\ell+1)-1-\mu(n-\ell-1) &= \mu(n^{1/\mu}\ell-n+\ell)+n^{1/\mu}+\mu-1 \\ &\le \mu(n^{1/\mu}cn-n+cn)+n^{1/\mu}+\mu-1 \\ &= \mu n(c(n^{1/\mu}+1)-1)+n^{1/\mu}+\mu-1 \quad .\end{aligned}$$

Table 8. Number of preprocessed tuples needed for basic operations

operation	tuple type	#tuples	ring	length
Linear combination	–	–	–	–
Conversion to a smaller ring	–	–	–	–
Bit decomposition in $\mathbb{Z}_m$	bit	$\log m$	m	–
Multiplication in $\mathbb{Z}_m$	triple	1	m	–
Extending $\mathbb{Z}_{m_x}$ to $\mathbb{Z}_{m_y}$	bit	$\log m_x$	m_y	–
Comparison in $\mathbb{Z}_m$	bit	$3\log m + 1$	m	–
Bit shift in $\mathbb{Z}_m$	cv	1	m	$\log m$
	triple	1	m	–
Rotation of length ℓ in $\mathbb{Z}_m$	rot	1	m	ℓ

If we take $c = 1/(n^{1/\mu} + 1)$, then this quantity is upper bounded by $n^{1/\mu} + \mu - 1$, which is a sufficient choice for κ.

Now let $\ell \geq cn$. In this case, by (2), already the probability of passing cut-and-choose is less than 2^{-ck}, on the condition $k \leq \frac{\mu n}{2}$. It is sufficient to take $k \geq \eta/c = \eta(n^{1/\mu} + 1)$ for this probability to be smaller than $2^{-\eta}$.

Due to the condition $k \leq \frac{\mu n}{2}$, we need to show that $\eta(n^{1/\mu} + 1) \leq \frac{\mu n}{2}$, and $n^{1/\mu} + \mu - 1 \leq \frac{\mu n}{2}$. We have shown that, for catching a single tuple, we should anyway choose $\mu \geq 1 + \eta/\log n$. We get

$$\eta(n^{1/\mu} + \mu - 1) \leq n^{1/(1+\eta/\log n)} + \mu - 1 \leq n^{\log n/\eta} + \mu - 1 = 2^{-\eta} + \mu - 1 \leq \mu \leq \frac{\mu n}{2}$$

for $n \geq 2$, and

$$\eta(n^{1/\mu} + 1) \leq \eta(n^{1/(1+\eta/\log n)} + 1) \leq \eta(n^{\log n/\eta} + 1) = \eta(2^{-\eta} + 1) \leq \frac{3}{2}\eta\ .$$

In order to get $\frac{\mu n}{2} \geq \frac{3\eta}{2}$, we need $\mu \geq \frac{3\eta}{n}$. Since $\mu \geq 1 + \eta/\log n > \eta/\log n$, it suffices to have $\log n \leq \frac{n}{3}$, which is true for $n \geq 12$. Such a choice for n is reasonable, since we may always generate more tuples than we actually need, and the preprocessing phase is in general run for several future protocol runs.

Summary. In order to allow a bad tuple pass with the probability of at most $2^{-\eta}$, while ending up with n tuples, it is sufficient to choose the parameters μ and κ as follows:

$$\begin{aligned}&\mu \geq 1 + \eta/\log n\\ &\kappa \geq \max\{(n^{1/\mu} + 1)\eta, n^{1/\mu} + \mu - 1\}\ .\end{aligned}$$

This choice of μ and κ is given for each type of tuples separately. If the total number of generated tuples is N, then it suffices in any case to take $\mu \geq 1 + \eta/\log N$ and $\kappa \geq \max\{(N^{1/\mu} + 1)\eta, N^{1/\mu} + \mu - 1\}$. $\square$

C.4 Verification of Linear Subcircuits

We now present a formal protocol Π_{verify} implementing the ideal functionality $\mathcal{F}_{verify}$ of Fig. 3. The protocol Π_{verify} is given in Fig. 17-20. It works on top of the sharing functionality $\mathcal{F}_{share}$ defined in Sec. C.2, and the preprocessed tuple generation functionality $\mathcal{F}_{pre}$ defined in Sec. C.3.

Observation 4 *We may extract the number of different tuples required for each operation type directly from the description of the initialization phase. They are given in Table 8.*

Lemma 7 (cost of initializing Π_{verify}). *Let Π_{verify} use the implementation Π_{pre} of $\mathcal{F}_{pre}$ with c-bit randomness seed, and the parameters μ, κ. Let all the functions f to be verified consist of basic operations f_i, such that there are N_b operations requiring bit decompositions (bit decomposition, ring extension, bit*

In Π_{verify}, each party works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$, the ring cardinality $m(id)$, the operation $f(id)$ to verify, the input identifiers $\boldsymbol{xid}(id)$, and the output identifiers $\boldsymbol{yid}(id)$ on which $f(id)$ should be verified. The prover stores the committed values in a local array *comm*. The verifiers store the helpful values published by the verifier in an array *pubv*. The messages that are not committed yet are stored by the sender and the receiver in a local array *sent*. Π_{verify} uses $\mathcal{F}_{transmit}$, $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$ as subroutines. The protocol uses a one-way hash function H.

Initialization: On input $(\mathsf{init}, \hat{m}, \hat{f}, \hat{\boldsymbol{xid}}, \hat{\boldsymbol{yid}}, \hat{p}, \hat{p'})$ from all the (honest) parties, where the domains of the input functions $\hat{m}, \hat{f}, \hat{\boldsymbol{xid}}, \hat{\boldsymbol{yid}}, \hat{p}, \hat{p'}$ are all the same, initialize *comm* and *sent* to empty arrays. Assign the mappings $m \leftarrow \hat{m}$, $f \leftarrow \hat{f}$, $\boldsymbol{xid} \leftarrow \hat{\boldsymbol{xid}}$, $\boldsymbol{yid} \leftarrow \hat{\boldsymbol{yid}}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$.

Initializing subroutine protocols:

- *Initialize* $\mathcal{F}_{transmit}$: For all $id \in \mathsf{Dom}(f)$, define $s(id) \leftarrow p(id)$, $r(id) = f(id) \leftarrow p'(id)$. For id corresponding to randomness, define also $s((id, j, k)) \leftarrow j$, $r((id, j, k)) \leftarrow p'(id)$, $f((id, j, k)) = k$. Send (init, s, r, f) to $\mathcal{F}_{transmit}$.
- *Initialize* $\mathcal{F}_{pre}$: A message $(\mathsf{init}, \bar{\ell}, \bar{m}, \bar{n}, \bar{p})$ is sent to $\mathcal{F}_{pre}$, where $\bar{\ell}$,$\bar{m}$,$\bar{n}$,$\bar{p}$ depend on the functions f to be verified. Let $f(id)$ be a composition of basic operations $f_1, \ldots, f_{N_{id}}$. Each such f_i, introduces to $\mathcal{F}_{pre}$ identifiers of the form $id' \leftarrow (id_i, type, \ell, m, n)$ such that $type$ is the type of the tuple, $\bar{\ell}(id') = \ell$, $\bar{m}(id') = m$, $\bar{n}(id') = n$. For all id', take $\bar{p}(id') \leftarrow p(id)$.
  1. *Linear combination, conversion to a smaller ring:* no tuples needed.
  2. *Bit decomposition in* $\mathbb{Z}_m$: $(id_i, \mathsf{bit}, 1, m, \log m)$;
  3. *Multiplication in* $\mathbb{Z}_m$*:* $(id_i, \mathsf{triple}, 1, m, 1)$;
  4. *Extending* $\mathbb{Z}_{m_x}$ *to a larger ring* $\mathbb{Z}_{m_y}$*:* $(id_i, \mathsf{bit}, 1, m_y, \log m_x)$;
  5. *Comparison in* $\mathbb{Z}_m$*:* $(id_i, \mathsf{bit}, 1, m+1, 3\log m + 1)$;
  6. *Bit shift in* $\mathbb{Z}_m$*:* $(id_i, \mathsf{cv}, \log m, m, 1)$, $(id_i, \mathsf{triple}, 1, m, 1)$;
  7. *Rotation of length* ℓ *in* $\mathbb{Z}_m$*:* $(id_i, \mathsf{rot}, \ell, m, 1)$.

  Let *pre* be the array containing all such identifiers introduced by all basic operations of $f(id)$. Since $\mathcal{F}_{pre}$ generates all the tuples of the same type simultaneously, we may optimize tuple generation by substituting all the identifiers $(id_i, type, \ell, m, n_{id_i})$ for the same $type$, m, ℓ with a single identifier $id' = (type, \ell, m, n)$ for $n = \sum n_{id_i}$. After inducing $\bar{\ell}, \bar{m}, \bar{n}, \bar{p}$ from these new identifiers, each (honest) party sends $(\mathsf{init}, \bar{\ell}, \bar{m}, \bar{n}, \bar{p})$ to $\mathcal{F}_{pre}$.
- *Initialize* $\mathcal{F}_{share}$: The identifiers should be reserved for the following values:
  1. Each of the tuples for which $\mathcal{F}_{pre}$ was initialized reserves a sharing for each its element, whose precise number depends on the type of the tuple. Define $id' \leftarrow id^{type}_{i,k}$, $\tilde{m}(id') \leftarrow m$, $\tilde{p}(id') \leftarrow \bar{p}(id')$, $\tilde{p'}(id') \leftarrow \bar{p'}(id')$.
  2. For sharing the inputs and the outputs of $f(id)$, take $\tilde{m}(id) \leftarrow m(id)$, $\tilde{p}(id) \leftarrow p(id)$, $\tilde{p'}(id) \leftarrow p'(id)$. If any such element is randomness, define additionally $\tilde{m}(id_j) \leftarrow m(id)$, $\tilde{p}(id_j) \leftarrow j$, and $\tilde{p'}(id_j) \leftarrow p(id)$.
  3. A special index id^m_1 is introduced to store the value 1 shared in $\mathbb{Z}_m$.

  Each (honest) party sends $(\mathsf{init}, \tilde{m}, \tilde{n}, \tilde{p}, \tilde{p}')$ is sent to $\mathcal{F}_{share}$.

Generating preshared triples: A message $(type, id')$ is sent to $\mathcal{F}_{pre}$ by each (honest) party for each id' on which $\mathcal{F}_{pre}$ was initialized. Upon getting $\boldsymbol{s}$ as a response for $(type, id')$ from $\mathcal{F}_{pre}$, each (honest) party reads the next n' tuples for each $(id_i, type, \ell, m, n') \in pre$, puts them all sequentially into a vector s_j, and sends $(\mathsf{reshare}, s_j, id^{type}_{i,j})$ to $\mathcal{F}_{share}$. $P_{p(id)}$ writes $comm[id^{type}_{i,j}] \leftarrow s_j$.

Fig. 17: Real protocol Π_{verify} (initialization)

Input Commitment: On input $(\mathsf{commit\_input}, x, id)$, $P_{p(id)}$ sends $(\mathsf{share}, (x^k)_{k\in[n]}, id)$ to $\mathcal{F}_{share}$. On input $(\mathsf{commit\_input}, id)$, each (honest) party sends (share, id) to $\mathcal{F}_{share}$.
Message Commitment:

1. On input $(\mathsf{send\_msg}, m, id)$, $P_{p(id)}$ sends $(\mathsf{transmit}, id, m)$ to $\mathcal{F}_{transmit}$.
2. On input $(\mathsf{commit\_msg}, m, id)$, $P_{p(id)}$ sends $(\mathsf{mshare}, (m^k)_{k\in[n]}, id)$ to $\mathcal{F}_{share}$. On input $(\mathsf{commit\_msg}, id)$, each (honest) party sends (mshare, id) to $\mathcal{F}_{share}$.

Randomness Commitment: On input $(\mathsf{commit\_rnd}, id)$, each (honest) party P_j, $j \neq p(id)$ (actually, a set of any t parties is sufficient):

1. Generates a random value $r_j \in \mathbb{Z}_{m(id)}$, shares it to $(r_j^k)_{k\in[n]}$, and sends $(\mathsf{transmit}, (id, j, k), H(r_j^k))$ to $\mathcal{F}_{transmit}$.
2. Sends $(\mathsf{mshare}, (r_j^k)_{k\in[n]}, id_j)$ to $\mathcal{F}_{share}$.
3. Sends $(\mathsf{lc}, [1, \dots, 1], (id_j)_{j\in[n], p(id)\neq j}, id)$ to $\mathcal{F}_{share}$.

Stoppings: At any time, when $\mathcal{F}_{transmit}$, $\mathcal{F}_{pre}$, or $\mathcal{F}_{share}$ outputs a message $(\mathsf{corrupt}, k)$, output $(\mathsf{corrupt}, k)$ to $\mathcal{Z}$. Treat P_k as if it has left the protocol. Assign $p(id) \leftarrow p'(id)$ for $p(id) = k$, and $p'(id) \leftarrow p(id)$ for $p'(id) = k$.

Fig. 18: Real protocol Π_{verify} (commitments, and stopping)

Verification: On input (verify, id), each (honest) party P_k decomposes $f(id)$ to basic operations $f_1, \dots, f_N$. For each f_i, some additional identifiers are defined: $id_i^{\mathsf{x}_k}$ for the k-th input, $id_i^{\mathsf{y}_k}$ for the k-th output, and $id_i^{\mathsf{z}_k}$ for the k-th alleged zero of f_i (some of these will actually overlap). The index k is omitted if there is only one such identifier. The symbols other than x, y, z will be used for intermediate values.
For shortness of notation, we write 1 instead of the identifier in which $\mathcal{F}_{share}$ stores the public value 1 over a certain ring of cardinality m (where m is omitted from the notation since it can be easily derived from the context).
First, $P_{p(id)}$ computes all the intermediate values $comm[id^i_{\mathsf{x}_k}]$. Let $\hat{\boldsymbol{x}} = [\hat{\boldsymbol{x}}_1 \| \cdots \| \hat{\boldsymbol{x}}_N]$ denote all values that should be broadcast by $P_{p(id)}$, where $\hat{\boldsymbol{x}}_i$ depends on f_i, the results $comm[id^i_{\mathsf{x}_k}]$, and the previously generated tuples $comm[id^{type}_{i,k}]$ as follows:

1. *Linear combination*: no broadcasts.
2. *Multiplication in $\mathbb{Z}_m$*: $\hat{\boldsymbol{x}}_i = [(x_1 - a) \bmod m, (x_2 - b) \bmod m]$ for $a \leftarrow comm[id^{\mathsf{triple}}_{i,1}]$, $b \leftarrow comm[id^{\mathsf{triple}}_{i,2}]$, $x_1 \leftarrow comm[id_i^{\mathsf{x}_1})$, $x_2 \leftarrow comm[id_i^{\mathsf{x}_2}]$.
3. *Bit decomposition in $\mathbb{Z}_m$*: $\hat{\boldsymbol{x}}_i = [c_1, \dots, c_{\log m}]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit $comm[id^{\mathsf{bit}}_{i,k}]$ is different from the k-th bit of $comm[id_i^{\mathsf{x}}]$.
4. *Conversion to a smaller ring*: no broadcasts.
5. *Conversion from $\mathbb{Z}_{m_x}$ to a larger ring $\mathbb{Z}_{m_y}$*: Perform bit decomposition of $comm[id_i^{\mathsf{x}}]$ over $\mathbb{Z}_{n_y}$, getting $\log n_y$ bits b_k. Take the first $\log n_x$ of these bits.
$\hat{\boldsymbol{x}}_i = [c_1, \dots, c_{\log n_x}]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit $comm[id^{\mathsf{bit}}_{i,k}]$ is different from b_k.
6. *Comparison*: Let $x_1 \leftarrow comm[id_i^{\mathsf{x}_1}]$, $x_2 \leftarrow comm[id_i^{\mathsf{x}_2}]$. Perform bit decomposition of x_1, x_2 and $x_1 - x_2$ over $\mathbb{Z}_{m+1}$. Take the first $\log m$ bits b_k, $b_{k+\log m}$ ($k \in [\log m]$) of the first two decompositions, and all the $\log m + 1$ bits $b_{k+2\log m}$ ($k \in [\log m + 1]$) of the third decomposition.
$\hat{\boldsymbol{x}}_i = [c_1, \dots, c_{3\log m + 1}]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit $comm[id^{\mathsf{bit}}_{i,k}]$ is different from b_k.
7. *Bit Shift*: Let $x_1 \leftarrow comm[id_i^{\mathsf{x}_1}]$, $x' \leftarrow comm[id_i^{\mathsf{x}_2}]$, $r \leftarrow comm[id^{\mathsf{cv}}_{i,1}]$, $a \leftarrow comm[id^{\mathsf{triple}}_{i,1}]$, $b \leftarrow comm[id^{\mathsf{triple}}_{i,2}]$. Compute $x_2 \leftarrow 2^{x'}$.
$\hat{\boldsymbol{x}}_i = [(x' - r) \bmod (\log m), (x_1 - a) \bmod m, (x_2 - b) \bmod m]$.
8. *Rotation of length ℓ in $\mathbb{Z}_m$*: $\hat{\boldsymbol{x}}_i = [[(x' - r) \bmod \ell] \| (\boldsymbol{x} - \boldsymbol{a}) \bmod m]$ for $r \leftarrow comm[id^{\mathsf{rot}}_{i,1}]$, $\boldsymbol{a} \leftarrow comm[id^{\mathsf{rot}}_{i,k+\ell+1}]]_{k\in[\ell]}$, $x' \leftarrow comm[id_i^{\mathsf{x}_2}]$, $\boldsymbol{x} \leftarrow comm[id_i^{\mathsf{x}_1}]$.

$P_{p(id)}$ sends $(\mathsf{broadcast}, \mathsf{public}, (id_i^{type}, \hat{\boldsymbol{x}}_i)_{i\in[N]})$ to $\mathcal{F}_{transmit}$. Upon receiving $(\mathsf{broadcast}, \mathsf{public}, (id_i^{type}, \hat{\boldsymbol{x}}_i)_{i\in[N]})$, each (honest) party writes $pubv[id_i^{type}] \leftarrow \hat{\boldsymbol{x}}_i$. For simplicity of further verifications, we assume that $(\mathsf{lc}, [1, -1], [1, id^{\mathsf{bit}}_{i,k}], id^{\mathsf{bit}}_{i,k})$ is immediately sent to $\mathcal{F}_{share}$ for all k such that $c_k = 1$ was broadcast, so that we do not need to compute it for each operation separately.

Fig. 19: Real protocol Π_{verify} (broadcast)

The verifier computation depends on f_i. They start jointly generating the values id_i^{y} (that they will use also as id_i^{x}), and the alleged zeroes id_i^{z}.

1. *Linear combination* $[c_1, \dots, c_m]$:
   Send $(\mathsf{lc}, [c_1, \dots, c_m], [id_i^{x_1}, \dots, id_i^{x_m}], id_i^{y})$ to $\mathcal{F}_{share}$.
2. *Multiplication in* $\mathbb{Z}_m$: Let $(id^a, id^b, id^c) \leftarrow (id_{i,k}^{\mathsf{triple}})_{k \in [3]}$, and $[\hat{x}_1, \hat{x}_2] \leftarrow pubv[id_i^{\mathsf{triple}}]$. Send to $\mathcal{F}_{share}$
   - $(\mathsf{lc}, [\hat{x}_1 \cdot \hat{x}_2, \hat{x}_1, \hat{x}_2, 1], [1, id^b, id^a, id^c], id_i^{\mathsf{y}})$;
   - $(\mathsf{lc}, [\hat{x}_1, 1, -1], [1, id^a, id_i^{\mathsf{x}_1}], id_i^{\mathsf{z}_1})$;
   - $(\mathsf{lc}, [\hat{x}_2, 1, -1], [1, id^b, id_i^{\mathsf{x}_2}], id_i^{\mathsf{z}_2})$.
3. *Bit decomposition in* $\mathbb{Z}_m$: Let $[id^{b_0}, \dots, id^{b_{\log m - 1}}] \leftarrow (id_{i,k-1}^{\mathsf{bit}})_{k \in [\log m]}$.
   Send $(\mathsf{lc}, [2^{k-1}]_{k \in [\log m]}, [id^{b_{k-1}}]_{k \in [\log m]}, id_i^{\mathsf{y}})$ to $\mathcal{F}_{share}$.
4. *Conversion to a smaller ring*: Send $(\mathsf{trunc}, id_i^{\mathsf{x}}, id_i^{\mathsf{y}})$ to $\mathcal{F}_{share}$.
5. *Conversion from* $\mathbb{Z}_{m_x}$ *to a larger ring* $\mathbb{Z}_{m_y}$: Let $[id^{b_0}, \dots, id^{b_{\log m_x - 1}}] \leftarrow (id_{i,k-1}^{\mathsf{bit}})_{k \in [\log m_x]}$. Send to $\mathcal{F}_{share}$
   - $(\mathsf{lc}, [2^{k-1}]_{k \in [\log m_x]}, [id^{b_{k-1}}]_{k \in [\log m_x]}, id_i^{\mathsf{y}})$;
   - $(\mathsf{trunc}, id_i^{\mathsf{y}}, id_i^{\mathsf{w}})$;
   - $(\mathsf{lc}, [1, -1], [id_i^{\mathsf{x}}, id_i^{\mathsf{w}}], id_i^{\mathsf{z}})$.
6. *Comparison*: Let $id^{b_0}, \dots, id^{b_{3 \log m}} \leftarrow (id_{i,k-1}^{\mathsf{bit}})_{k \in [3 \log m + 1]}$. Repeat the steps of (5), converting the inputs x_1 and x_2 from $\mathbb{Z}_m$ to $\mathbb{Z}_{m+1}$, using up the first $2 \log m$ bits id^{b_k}. Let the results be written into $id_i^{\mathsf{y}_1}$, $id_i^{\mathsf{y}_2}$, and the alleged zeroes into $id_i^{\mathsf{z}_1}$, $id_i^{\mathsf{z}_2}$ respectively. Send to $\mathcal{F}_{share}$
   - $(\mathsf{lc}, [2^{k-1}]_{k \in [\log m + 1]}, [id^{b_{2 \log m + k - 1}}]_{k \in [\log m + 1]}, id_i^{\mathsf{w}})$;
   - $(\mathsf{lc}, [1, -1, -1], [id_i^{\mathsf{y}_1}, id_i^{\mathsf{y}_2}, id_i^{w}], id_i^{\mathsf{z}_3})$;
   - Take $id_i^{\mathsf{y}} \leftarrow id^{b_{3m}}$.
7. *Bit Shift*: Let $[\hat{r}] \leftarrow pubv[id_i^{\mathsf{cv}}]$. Let $(id^r, id^{s_1}, \dots, id^{s_{\log m}}) \leftarrow [id_{i,1}^{\mathsf{cv}}, id_{i,2}^{\mathsf{cv}}, \dots, id_{i,\log m+1}^{\mathsf{cv}}]$. Send to $\mathcal{F}_{share}$:
   - $(\mathsf{lc}, [2^{k-1}]_{k \in [\log m]}, [id^{s_{\hat{r}+k-1}}]_{k \in [\log m]}, id_i^{\mathsf{w}})$;
   - $(\mathsf{lc}, [1, -1, -\hat{r}], [id_i^{\mathsf{x}_1}, id^r, 1], id_i^{\mathsf{z}_1})$.

   Repeat the steps of (2) multiplying id_i^{x} and id_i^{w}, writing the result into id_i^{y} and obtaining alleged zeroes $id_i^{\mathsf{z}_2}$ and $id_i^{\mathsf{z}_3}$.
8. *Rotation*: Let $id^r \leftarrow id_{i,1}^{\mathsf{rot}}$, $id^{s_k} \leftarrow id_{i,k+1}^{\mathsf{rot}}$, $id^{a_k} \leftarrow id_{i,k+\ell+1}^{\mathsf{rot}}$, $id^{b_k} \leftarrow id_{i,k+2\ell+1}^{\mathsf{rot}}$ for $k \in [\ell]$. Let $[\hat{r}, \hat{\boldsymbol{x}}] \leftarrow pubv[id_i^{\mathsf{rot}}]$.
   Send to $\mathcal{F}_{share}$:
   - $(\mathsf{lc}, \hat{\boldsymbol{x}}, [id^{s_{k+j-1}}]_{j \in \ell}, id_i^{\mathsf{c}_k})$ for all $k \in \ell$;
   - $(\mathsf{lc}, [1, 1], [id_i^{\mathsf{c}_{\hat{r}+k-1}}, id^{b_{\hat{r}+k-1}}], id_i^{\mathsf{y}_k})$ for $k \in \ell$;
   - $(\mathsf{lc}, [1, -1, -\hat{r}], [id_i^{\mathsf{x}_1}, id^r, 1], id_i^{\mathsf{z}_1})$;
   - $(\mathsf{lc}, [1, -1, -\hat{x}_k], [id_i^{\mathsf{x}_{k+1}}, id^{a_{k+1}}, 1], id_i^{\mathsf{z}_{k+1}})$ for $k \in \ell$.

If $i = N$, then each (honest) party also sends $(\mathsf{lc}, [1, -1], [id_N^{\mathsf{y}_k}, id_k], id_{N+1}^{\mathsf{z}_k})$ to $\mathcal{F}_{share}$ for all $id_k \leftarrow \boldsymbol{yid}_k$.
After all f_i have been processed, for each obtained $id_i^{\mathsf{z}_k}$, each (honest) party first sends $(\mathsf{weak\_open}, id_i^{\mathsf{z}_k})$ to $\mathcal{F}_{share}$. If $\mathcal{F}_{share}$ outputs $\perp$, then each (honest) party sends $(\mathsf{open}, id_i^{\mathsf{z}_k})$ to $\mathcal{F}_{share}$. Upon receiving all $(id_i^{\mathsf{z}_k}, z_{ik})$ from $\mathcal{F}_{share}$ (or all the values x_{ik} needed to compute z_{ik}), if $z_{ik} = 0$ for all i, k, then an honest party outputs 1. Otherwise, it outputs 0.

Fig. 20: Real protocol $\mathit{\Pi}_{verify}$ (verifying operations)

shift, comparison), and N_r other non-linear operations (rotating a vector of length ℓ is treated as ℓ different operations). Let m be the cardinality of the largest ring involved in the computation, and let $\mathsf{m} := \log m$. The cost of initializing Π_{verify} is upper bounded by $t \cdot \mathsf{bc}_c + t \cdot \mathsf{bc}_c + n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot ((3\mathsf{m}+1)(\mu(N_b \cdot \mathsf{m}+N_r)+\kappa(\mathsf{m}+1)))} + \mathsf{bc}_{\mathsf{sh}_n \cdot (\kappa(3\mathsf{m}+1)(\mathsf{m}+1))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot \mathsf{m}+2 \cdot N_r))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot (3\mathsf{m}+1)+2 \cdot N_r))}$.

Proof. In Table 6, the reshare functionality of Π_{share} does not have any cost. Hence the cost comes only from the generation of preprocessed tuples. The number of different tuples used by each operation is given in Table. 8. By Lemma 5, the cost of generating N tuples of type x of length ℓ over ring of size m is $t \cdot \mathsf{bc}_c + t \cdot \mathsf{bc}_c + n \cdot \mathsf{tr}_{(\mu N+\kappa) \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell))} + \mathsf{bc}_{\kappa \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell))} + \mathsf{bc}_{(\mu-1)N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_1}(x,m,\ell))} + \mathsf{bc}_{(\mu-1)N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_2}(x,m,\ell))}$ (the definitions of subterms can be found in Table 7). In this way, the total number of the transmitted and broadcast bits is linear in the terms $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell))$, $N \cdot \mathsf{nbits}_{\mathsf{open}_1}(x,m,\ell)$, and $N \cdot \mathsf{nbits}_{\mathsf{open}_2}(x,m,\ell)$. Hence it suffices to find the upper bounds for these three quantities.

– For the bit-related gates, the largest value for $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell))$ is $N_b \cdot (3 \log m + 1) \cdot \mathsf{nbits}_{\mathsf{tuple}}(\mathsf{bit}, m, 1) = N_b \cdot (3 \log m + 1) \cdot \mathsf{sh}_n \cdot (\log m)$, which comes from the comparison gate. Similarly, $N_b \cdot (3 \log m + 1) \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_2}(\mathsf{bit}, m, 1)) = N_b \cdot (3 \log m + 1) \cdot \mathsf{sh}_n \cdot (\log m)$ is the largest value for $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_2}(x,m,\ell))$. For $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_1}(x,m,\ell))$, the largest value $N_b \cdot 1 \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_1}(\mathsf{cv}, m, \log m)) = N_b \cdot \mathsf{sh}_n \cdot (\log^2 m)$ comes from the bit shift gate.
– For the multiplication and rotation gates, the largest value for $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(x,m,\ell))$ term is $N_r \cdot 1 \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(\mathsf{rot}, m, 1)) = N_r \cdot \mathsf{sh}_n \cdot (3 \log m + 1)$, which comes from the rotation gate unit. Similarly, $N_r \cdot 1 \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{tuple}}(\mathsf{rot}, m, 1)) = N_r \cdot \mathsf{sh}_n \cdot (3 \log m + 1)$ is the largest value for $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_2}(x,m,\ell))$. For $N \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_1}(x,m,\ell))$, the largest value $N_r \cdot 1 \cdot \mathsf{sh}_n \cdot (\mathsf{nbits}_{\mathsf{open}_1}(\mathsf{triple}, m, \log m)) = N_r \cdot \mathsf{sh}_n \cdot (2 \log m)$ comes from the multiplication gate.

The randomness seed of c bits may be generated once for all the tuples. Summing up the upper bounds for each operation, and assuming that all of them can be transmitted and broadcast in parallel, we may sum up the total number of bits in each round. For simplicity, let μ and κ be the same for generating all types of tuples. For the shortness of notation, let $\mathsf{m} := \log m$. Let x_1, x_2, x_3, x_4 be the total number of bits transmitted/broadcast in the given 4 rounds, i.e such that the total cost is $t \cdot \mathsf{bc}_c + n \cdot \mathsf{tr}_{x_1} + \mathsf{bc}_{x_2} + \mathsf{bc}_{x_2} + \mathsf{bc}_{x_4}$. We use the fact that the share overhead is linear w.r.t the number of shared bits, i.e $\mathsf{sh}_n \cdot (M_1 + M_2) = \mathsf{sh}_n \cdot M_1 + \mathsf{sh}_n \cdot M_2$ (see Observation 2).

$$
\begin{aligned}
x_1 &\le (\mu N_b + \kappa)(3\mathsf{m}+1)\mathsf{sh}_n \cdot (\mathsf{m}) + (\mu N_r + \kappa)\mathsf{sh}_n \cdot (3\mathsf{m}+1) \\
&= \mathsf{sh}_n \cdot (\mu N_b(3\mathsf{m}+1)\mathsf{m} + \mu N_r(3\mathsf{m}+1) + \kappa(3\mathsf{m}+1)(\mathsf{m}+1)) \\
&= \mathsf{sh}_n \cdot (\mu(3\mathsf{m}+1)(N_b\mathsf{m} + N_r) + \kappa(3\mathsf{m}+1)(\mathsf{m}+1)) \\
&= \mathsf{sh}_n \cdot ((3\mathsf{m}+1)(\mu(N_b\mathsf{m}+N_r) + \kappa(\mathsf{m}+1))) \ . \\
x_2 &\le \kappa(3\mathsf{m}+1)\mathsf{sh}_n \cdot (\mathsf{m}) + \mathsf{sh}_n \cdot (3\mathsf{m}+1) \\
&= \mathsf{sh}_n \cdot (\kappa(3\mathsf{m}+1)(\mathsf{m}+1)) \ . \\
x_3 &\le (\mu-1)(N_b \cdot \mathsf{sh}_n \cdot (\mathsf{m}^2) + N_r \cdot \mathsf{sh}_n \cdot (2\mathsf{m})) \\
&= \mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b\mathsf{m} + 2N_r)) \ . \\
x_4 &\le (\mu-1)(N_b(3\mathsf{m}+1)\mathsf{sh}_n \cdot (\mathsf{m}) + N_r \cdot \mathsf{sh}_n \cdot (2\mathsf{m})) \\
&= \mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b(3\mathsf{m}+1) + 2N_r)) \ .
\end{aligned}
$$

The total cost is bounded by $t \cdot \mathsf{bc}_c + t \cdot \mathsf{bc}_c + n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot ((3\mathsf{m}+1)(\mu(N_b \cdot \mathsf{m}+N_r)+\kappa(\mathsf{m}+1)))} + \mathsf{bc}_{\mathsf{sh}_n \cdot (\kappa(3\mathsf{m}+1)(\mathsf{m}+1))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot \mathsf{m}+2 \cdot N_r))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot (3\mathsf{m}+1)+2 \cdot N_r))}$. □

Observation 5 *From the description of the commitment functions of Π_{verify}, we may count the number of $\mathcal{F}_{transmit}$ and $\mathcal{F}_{share}$ calls that it makes. They are given in Table 9*

Lemma 8 (cost of the commitments of Π_{verify}). *Let $\mathcal{F}_{verify}$ use the implementation Π_{share} of $\mathcal{F}_{share}$. Let N_x be the number of inputs, N_r the number of random elements, an N_c the number of message elements*

Table 9. Number of $\mathcal{F}_{share}$ operations needed for commitments of N-bit values in $\mathcal{F}_{verify}$

commitment type	called functionality	#bits	#calls
commit_input	share	N	1
send_msg	transmit	N	1
commit_msg	mshare	N	1
commit_rnd	transmit	c	nt
	mshare	N	t

Table 10. Number of bits broadcast by the prover for each operation in $\mathcal{F}_{verify}$

operation	#bits
Linear combination, conversion to a smaller ring	0
Bit decomposition in $\mathbb{Z}_m$	$\log m$
Multiplication in $\mathbb{Z}_m$:	$2 \log m$
Extending $\mathbb{Z}_{m_x}$ to $\mathbb{Z}_{m_y}$	$\log m_x$
Comparison in $\mathbb{Z}_m$	$3 \cdot \log m + 1$
Bit shift in $\mathbb{Z}_m$	$\log\log m + 2\log m$
Rotation of length ℓ in $\mathbb{Z}_m$	$\log \ell + \ell \log m$

over a ring of size m. Let $\mathsf{m} := \log m$ Taking into account the costs of different operations of Π_{share} given in Table 6, the total number of $\mathcal{F}_{transmit}$ operations needed to commit these elements is $nN_x \cdot \mathsf{tr}_{\mathsf{sh}_n\cdot(\mathsf{m})} + (ntN_r \cdot \mathsf{tr}_c + ntN_r \cdot \mathsf{tr}_{\mathsf{sh}_n\cdot(\mathsf{m})} + ntN_r \cdot \mathsf{fwd}_{\mathsf{sh}_n\cdot(\mathsf{m})}) + (N_c \cdot \mathsf{tr}_{\mathsf{sh}_n\cdot(\mathsf{m})} + nN_c \cdot \mathsf{tr}_{\mathsf{sh}_n\cdot(\mathsf{m})} + nN_c \cdot \mathsf{fwd}_{\mathsf{sh}_n\cdot(\mathsf{m})})$.

Proof. For each of the N_x inputs, commit_input is called. For each of the N_r random elements, commit_rnd is called. For each of the N_c message elements, send_msg and commit_msg are called sequentially. The definition of Π_{verify} does not state whether the inputs, the randomness, and the messages are committed in parallel, so the quantities N_x, N_r, N_c cannot be moved into subindices of tr and fwd without additional context. The final quantity is obtained by combining the values of Table 6 and Table 9. □

Observation 6 *By simply counting the number of broadcast bits for each basic operation, we get the numbers given in Table. 10. Note that the bits c_i broadcast for each bit decomposition do not have to be shared in $\mathbb{Z}_m$, and each such bit is broadcast as a 1-bit value.*

Lemma 9 (cost of the broadcasts of $\mathcal{F}_{verify}$). *Let all the functions f to be verified consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}\}$, treating a rotation of length ℓ as ℓ different operations. Let m be the size of the largest used ring, $\mathsf{m} := \log m$. The total cost of the broadcast phase of Π_{verify} is upper bounded by $\mathsf{bc}_{N\cdot(3\mathsf{m}+1)}$.*

Proof. All the bits are broadcast in parallel using $\mathcal{F}_{transmit}$. We use Table 10 to count the number of bits for each operation. We take the upper bound $3\log m + 1$ on broadcast bits per operation, which comes from the comparison operation. Differently from the initialization phase of Π_{verify}, the costs are similar for distinct types of basic operations, as they are all $O(\log m)$. □

Observation 7 *By simply counting the number of bits of the alleged zeroes for each operation, we get the results given in Table 11.*

Lemma 10 (cost of the final verification of Π_{verify}). *Let all the functions f to be verified consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}, \mathsf{bd}\}$, treating a rotation of length ℓ as ℓ different operations. Let M_y be the total number of bits output by f. Let m be the size of the largest used ring, $\mathsf{m} := \log m$. The cost of the verification phase of Π_{verify} is upper bounded by:*

- $n \cdot \mathsf{bc}_{\mathsf{sh}_n\cdot(N\cdot(3\mathsf{m}+1)+M_y)}$, *if* weak_open *succeeds.*
- $n \cdot \mathsf{rev}_{\mathsf{sh}_n\cdot(N\cdot(3\mathsf{m}+1)+M_y)}$, *if* weak_open *outputs* $\bot$.

Table 11. Number of alleged zero bits for verifying each operation in $\mathcal{F}_{verify}$

operation	#bits
Linear combination, conversion to a smaller ring	0
Bit decomposition in $\mathbb{Z}_m$	0
Multiplication in $\mathbb{Z}_m$:	$2 \log m$
Extending $\mathbb{Z}_{m_x}$ to $\mathbb{Z}_{m_y}$	$\log m_x$
Comparison in $\mathbb{Z}_m$	$\log (m+1) + 2 \log m$
Bit shift in $\mathbb{Z}_m$	$\log \log m + 2 \log m$
Rotation of length ℓ in $\mathbb{Z}_m$	$\log \ell + \ell \log m$

Proof. Taking into account the costs of different operations of Π_{share} given in Obs. 2, the functionalities lc and trunc do not take any communication. Hence the only cost for verifying different basic operations comes in the end, where the alleged zeroes are verified.

– Assume that weak_open succeeds for all alleged zeroes. It has cost $n \cdot \mathsf{bc}_m$ for an m-bit value. From Table. 11, we see that the largest number of alleged zero checks per operation (treating rotation as ℓ distinct operations) is $3\mathsf{m} + 1$, which comes from comparison. In addition, there is an alleged zero bit for each of the M_y output bits of f. The broadcast is parallelizable, so all the bits are broadcast simultaneously.
– Assume that weak_open returns $\perp$. In this case, open is used instead. Exactly the same values are revealed, but the underlying $\mathcal{F}_{transmit}$ operation is different. $\square$

Lemma 11. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{verify} UC-realizes $\mathcal{F}_{verify}$ in $\mathcal{F}_{transmit}$-$\mathcal{F}_{share}$-$\mathcal{F}_{pre}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{verify}$ described in Fig. 21. The simulator runs a local copy of Π_{verify}, together with local copies of $\mathcal{F}_{transmit}$, $\mathcal{F}_{share}$, $\mathcal{F}_{pre}$.

Simulatability During the commitments, $\mathcal{S}$ simulates $\mathcal{F}_{share}$, the inputs of dishonest parties for which are provided by $\mathcal{A}$. When the verification starts, $\mathcal{S}$ needs to simulate the broadcast, and it needs to generate the broadcast values of the honest provers itself. All of these values are some private values hidden by a random mask (each tuple is used only once), and hence are distributed uniformly. We discuss it in details for different kinds of tuples.

1. *Bit decomposition of x in $\mathbb{Z}_m$*: Since each b_k is distributed uniformly in $\mathbb{Z}_2$, the difference $b_k - x_k$ is also distributed uniformly in $\mathbb{Z}_2$.
2. *Multiplication of x_1 and x_2 in $\mathbb{Z}_m$*: Since the entries a and b of the triple (a, b, c) are distributed uniformly in $\mathbb{Z}_m$, so are the values $(x_1 - a) \bmod m$ and $(x_2 - b) \bmod m$.
3. *Characteristic Vector of x in $\mathbb{Z}_m$*: Since r is distributed uniformly in $\mathbb{Z}_\ell$, so is $(x - r) \bmod \ell$.
4. *Rotation of $\boldsymbol{x}$ of length ℓ in $\mathbb{Z}_m$ by x' positions*: Since r is distributed uniformly in $\mathbb{Z}_\ell$, and $\boldsymbol{a}$ in $\mathbb{Z}_m^\ell$, so are $(x - r) \bmod \ell$ and $(\boldsymbol{x} - \boldsymbol{a}) \bmod m$.

After all the broadcasts are simulated, $\mathcal{S}$ simulates opening to each party the alleged zero vector $\boldsymbol{z}$. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ already knows all the values needed to compute $\boldsymbol{z}$. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ obtains only the difference $f(\boldsymbol{x}) - \boldsymbol{y}$ from $\mathcal{F}_{verify}$. However, it needs to simulate the alleged zeroes $\boldsymbol{z}_i$ of *each* intermediate basic function f_i. Here we use the fact that, if $p(id) \notin \mathcal{C}$, then it has broadcast $\hat{\boldsymbol{x}}$ that indeed corresponds to the computation of $f(\boldsymbol{x})$. The only non-zero entries of $\boldsymbol{z}$ may come due to the mismatches between $f(\boldsymbol{x})$ and $\boldsymbol{y}$, and these differences $f(\boldsymbol{x}) - \boldsymbol{y}$ are provided by $\mathcal{F}_{verify}$.

Correctness The inputs, the messages, and the randomness are shared among the n parties by definition of $\mathcal{F}_{share}$. In addition, the preprocessed tuples are shared among the n parties by definition of $\mathcal{F}_{pre}$. The functionality reshare of $\mathcal{F}_{share}$ is used to put these shares together and allow to further use $\mathcal{F}_{share}$ as blackbox, doing computation on all these shares. For $p(id) \notin \mathcal{C}$, $\mathcal{F}_{share}$ takes the shares $(x^k)_{k \in [n]}$ provided by $P_{p(id)}$

Initialization: $\mathcal{S}$ calls $\mathcal{F}_{pre}$ to simulate the preprocessing phase. It gets all the n shares of dishonest provers, and up to $t-1$ shares of honest provers.
Input Commitment: $\mathcal{S}$ models sending $(\mathsf{share}, (x^k)_{k\in[n]}, id)$ and (share, id) to $\mathcal{F}_{share}$.
Message Commitment: On input $(\mathsf{send\_msg}, m, id)$, $\mathcal{S}$ models sending $(\mathsf{transmit}, id, m)$ to $\mathcal{F}_{transmit}$. On input $(\mathsf{commit\_msg}, m, id)$, $\mathcal{S}$ models sending $(\mathsf{mshare}, (m^k)_{k\in[n]}, id)$ and (mshare, id) to $\mathcal{F}_{share}$.
Randomness Commitment: On input $(\mathsf{commit\_rnd}, id)$, $\mathcal{S}$ needs to show to $\mathcal{A}$ the hashes generated by the honest parties. Since S has all information about $\mathcal{A}$ and the randomness that $\mathcal{A}$ uses, it knows the shares r_j for $j \in \mathcal{C}$ that are chosen by $\mathcal{A}$. Hence $\mathcal{S}$ generates $r_j \overset{\$}{\leftarrow} \mathbb{Z}_{m(id)}$ for $j \notin \mathcal{C}$ in such a way that $\sum_{j\in[n]} r_j$ equals to the appropriate value. $\mathcal{S}$ models sending $(\mathsf{transmit}, (id, j, k), H(r_j^k))$ to $\mathcal{F}_{transmit}$ and then $(\mathsf{mshare}, (r_j^k)_{k\in[n]}, id_j)$ and $(\mathsf{lc}, [1, \ldots, 1], (id_j)_{p(id)\neq j\in[n]}, id)$ to $\mathcal{F}_{share}$. Assuming that the outputs of H are indistinguishable from random, the behaviour of adaptive $\mathcal{A}$ does not depend on whether $\mathcal{S}$ has chosen truly random r_j for $j \notin \mathcal{C}$, or made them dependent on the values of r_j for $j \in \mathcal{C}$ that $\mathcal{A}$ would generate in the case if r_j for $j \notin \mathcal{C}$ were truly random.
Stoppings: At any time, when $\mathcal{F}_{transmit}$, $\mathcal{F}_{pre}$, or $\mathcal{F}_{share}$ should output a message $(\mathsf{corrupt}, k)$, $\mathcal{S}$ outputs $(\mathsf{corrupt}, k)$ to $\mathcal{F}_{verify}$. $\mathcal{S}$ discards P_k from its local run of Π_{verify}.
Verification: On input (verify, id), $\mathcal{S}$ decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$, and defines the additional identifiers $id_i^{\mathsf{x}k}$, $id_i^{\mathsf{y}k}$, $id_i^{\mathsf{z}k}$ as the honest parties do. For $p(id) \in \mathcal{C}$ It computes all the intermediate values $comm[id_i^{\mathsf{x}k}]$ and $comm[id_i^{\mathsf{y}k}]$, and broadcasts the values $\hat{\boldsymbol{x}}$ chosen by $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, broadcasting $\hat{\boldsymbol{x}}$ is to be simulated by $\mathcal{S}$ as follows (we use case distinction on types of preprocessed tuples causing the broadcast):

1. *Bit decomposition of x in $\mathbb{Z}_m$*: Need to broadcast $\hat{\boldsymbol{x}}_i = [c_1, \ldots, c_m]$, where $c_k \in \{0,1\}$ denotes whether the preshared trusted bit b_k is different from the k-th bit of x. Generate $c_k \overset{\$}{\leftarrow} \{0,1\}$.
2. *Multiplication of x_1 and x_2 in $\mathbb{Z}_m$*: Need to broadcast $\hat{\boldsymbol{x}}_i = [(x_1 - a) \bmod m, (x_2 - b) \bmod m]$ for the preshared multiplication triple (a, b, c). Generate $\hat{\boldsymbol{x}}_i \overset{\$}{\leftarrow} \mathbb{Z}_m^2$.
3. *Characteristic Vector of x in $\mathbb{Z}_m$*: Need to broadcast $\hat{\boldsymbol{x}}_i = [(x - r) \bmod \ell]$ for the preshared CV pair $(r, \boldsymbol{s})$. Generate $\hat{\boldsymbol{x}}_i \overset{\$}{\leftarrow} \mathbb{Z}_m^1$.
4. *Rotation of $\boldsymbol{x}$ of length ℓ in $\mathbb{Z}_m$ by x' positions*: Need to broadcast $\hat{\boldsymbol{x}}_i = [[(x' - r) \bmod \ell] \| (\boldsymbol{x} - \boldsymbol{a}) \bmod m]$ for the rotation tuple $(r, \boldsymbol{s}, \boldsymbol{a}, \boldsymbol{b})$. Generate $\hat{\boldsymbol{x}}_i \overset{\$}{\leftarrow} \mathbb{Z}_\ell^1 \times \mathbb{Z}_m^\ell$.

$\mathcal{S}$ simulates $(\mathsf{broadcast}, \mathsf{public}, (id_i^{type}, \hat{\boldsymbol{x}}_i)_{i\in[N]})$ using $\mathcal{F}_{transmit}$. Upon receiving $(\mathsf{broadcast}, \mathsf{public}, (id_i^{type}, \hat{\boldsymbol{x}}_i)_{i\in[N]})$, it writes $pubv[id_i^{type}] \leftarrow \hat{\boldsymbol{x}}_i$ for all honest parties, and sends the corresponding messages $(\mathsf{lc}, [1, -1], [1, id_{i,k}^{\mathsf{bit}}], id_{i,k}^{\mathsf{bit}})$ to $\mathcal{F}_{share}$, as the honest parties do.
The further computation depends on f_i, and S just sends to $\mathcal{F}_{share}$ the same messages that the honest parties send.

Fig. 21: The simulator $\mathcal{S}_{verify}$

itself. By definition of Π_{pre}, they are consistent with the shares of honest parties. It remains to prove that, if all these values are shared properly, then Π_{verify} does verify the computation of $f(id)$ on input (verify, id).

It easy to see that, if $\boldsymbol{z}_i = \boldsymbol{0}$ for the alleged zeroes produced by the basic function f_i, then f_i has been computed correctly with respect to the committed inputs and outputs on which it was verified, and $\hat{\boldsymbol{x}}_i$ has been computed correctly for f_i. The details of verifying each basic function are analogous to the preprocessed tuple generation proof of Lemma 6, and we do not repeat them here. If all f_i have been computed correctly, then so is the composition of f. □

C.5 The Main Protocol for Verifying SMC

The protocol Π_{vmpc} implementing $\mathcal{F}_{vmpc}$ is given in Fig. 22. It is built on top if the functionality $\mathcal{F}_{verify}$, used to verify the computation of each output of each round, with respect to the committed inputs, messages, and randomness.

Lemma 12. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{vmpc} UC-realizes $\mathcal{F}_{vmpc}$ in $\mathcal{F}_{verify}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{vmpc}$ described in Fig. 23. The simulator runs a local copy of Π_{vmpc}, together with a local copy of $\mathcal{F}_{verify}$.

• **In the beginning**, Each party P_i gets the message $(\mathsf{circuits}, i, (C^{\ell}_{ij})^{n,n,r}_{i,j,\ell=1,1,1})$ from $\mathcal{Z}$.

1. *Initializing $\mathcal{F}_{verify}$:* Let the n^{ℓ}_{ij} output wires of the circuit C^{ℓ}_{ij} be enumerated. For all $k \in [n^{\ell}_{ij}]$, the value $id \leftarrow (i, j, \ell, k)$ serves as an identifier for $\mathcal{F}_{verify}$. In addition, for each party P_i, there are identifiers (i, x, k) and (i, r, k) for the enumerated inputs and randomness respectively.
   – For each input wire $id \leftarrow (i, \mathsf{x}, k)$ or $id \leftarrow (i, \mathsf{r}, k)$, define $m(id)$ to be the ring in which the wire is defined, $f(id) \leftarrow \bot$, $\boldsymbol{xid}(id) \leftarrow \bot$, $\boldsymbol{yid}(id) \leftarrow \bot$, $p(id) = p'(id) = i$.
   – For each output wire $id \leftarrow (i, j, \ell, k)$, define $m(id)$ to be the ring in which the wire is defined, $f(id)$ a function consisting of basic operations of Sec. 4.3, computing the k-th coordinate of $\boldsymbol{m}^{\ell}_{ij} \leftarrow C^{\ell}_{ij}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell-1}_{ni})$ (this is always possible since every gate of C^{ℓ}_{ij} is by definition some basic operation), $\boldsymbol{xid}(id)$ the vector of all the identifiers of $\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell-1}_{ni}$ that are actually used by C^{ℓ}_{ij}, $\boldsymbol{yid}(id) \leftarrow [id]$, $p(id) = i$, $p'(id) = j$.

   Each (honest) party sends $(\mathsf{init}, m, f, \boldsymbol{xid}, \boldsymbol{yid}, p, p')$ to $\mathcal{F}_{verify}$.
2. *Randomness generation:* For each randomness input wire $id \leftarrow (i, \mathsf{r}, k)$, each (honest) party sends $(\mathsf{commit\_rnd}, id)$ to $\mathcal{F}_{verify}$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, P_i sends $(\mathsf{commit\_input}, \boldsymbol{x}_i, id)$ to $\mathcal{F}_{verify}$, and each other (honest) party sends $(\mathsf{commit\_input}, id)$ to $\mathcal{F}_{verify}$.

• **For each round** $\ell \in [r]$, P_i computes $\boldsymbol{m}^{\ell}_{ij} = C^{\ell}_{ij}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell-1}_{ni})$ for all $j \in [n]$, and sends $(\mathsf{send\_msg}, m^{\ell}_{ijk}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $k \in [|\boldsymbol{m}^{\ell}_{ij}|]$.
• **After** r **rounds**, each (honest) party P_i outputs $(\mathsf{output}, \boldsymbol{m}^r_{1i}, \ldots, \boldsymbol{m}^r_{ni})$ to $\mathcal{Z}$. Let $r' = r$ and $mlc_i[k] \leftarrow 0$ for all $k \in [n]$.
Alternatively, **at any time** before outputs are delivered to parties, if a message $(\mathsf{corrupt}, k)$ comes from $\mathcal{F}_{verify}$, each (honest) party P_i writes $mlc_i[k] \leftarrow 1$. In this case the outputs are not sent to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.
• **After** r' **rounds**:

1. Each (honest) party sends to $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n]$, $\ell \in [r']$, $k \in n^{\ell}_{ij}$.
2. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, each (honest) party sends (verify, id) to $\mathcal{F}_{verify}$, getting a vector of alleged zeroes $\boldsymbol{z}$ from $\mathcal{F}_{verify}$. If $\boldsymbol{z} = \boldsymbol{0}$, each (honest) party writes $mlc_i[k] \leftarrow 0$. Otherwise, it writes $mlc_i[k] \leftarrow 1$.

• **Finally**, each (honest) P_i outputs to $\mathcal{Z}$ the set of parties $\mathcal{B}_i$ such that $mlc_i[k] = 1$ iff $k \in \mathcal{B}_i$.

Fig. 22: The protocol Π_{vmpc} for verifiable computations

Simulatability $\mathcal{S}$ needs to simulate the messages $\boldsymbol{m}^{\ell}_{ij}$ that are computed by the honest parties P_i for corrupted parties P_j. It gets all such messages from $\mathcal{F}_{vmpc}$.

Another thing that should be simulated is the side-effect of $\mathcal{F}_{verify}$ that outputs the difference between the actual output of $f(\boldsymbol{x})$ and the output $\boldsymbol{y}$ that was committed by the prover. All the verifiable functions f of $\mathcal{F}_{verify}$ correspond to the computation of some output of a circuit C^{ℓ}_{ij} with respect to the committed inputs, randomness, and messages. By definition of $\mathcal{F}_{verify}$, unless at least one message $(\mathsf{corrupt}, p(id))$ has been output to each honest party (in this case $p(id) \in \mathcal{C}$), all these values are indeed committed as chosen by the party committing to them. Since each honest party has followed the protocol and computed C^{ℓ}_{ij} properly, and all its commitments are valid, the difference $f(\boldsymbol{x}) - \boldsymbol{y}$ should be 0 for honest parties, and so it is easy to simulate.

Correctness We need to prove that $\mathcal{F}_{verify}$ outputs exactly the same values as the parties in Π_{verify} would. By definition of $\mathcal{F}_{verify}$, there are two kinds of outputs:

1. *The computation output* $(\mathsf{output}, \boldsymbol{m}^r_{1i}, \ldots, \boldsymbol{m}^r_{ni})$. Let ℓ be any round. We prove by induction that each message $\boldsymbol{m}^{\ell}_{ij}$ seen by the adversary is consistent the $\mathcal{F}_{vmpc}$'s internal state.
   – *Base:* Initially, there are the inputs $\boldsymbol{x}_i$ and the randomness $\boldsymbol{r}_i$ in the internal state of $\mathcal{F}_{vmpc}$. So far, for $i \notin \mathcal{C}$, $\mathcal{A}$ has no information about $\boldsymbol{x}_i$, $\boldsymbol{r}_i$, and for $i \in \mathcal{C}$ it expects $\boldsymbol{x}_i = \boldsymbol{x}^*_i$, $\boldsymbol{r}_i = \boldsymbol{r}^*_i$, where $\boldsymbol{x}^*_i$ is chosen by $\mathcal{A}$ itself, and $\boldsymbol{r}^*_i$ is a uniformly distributed value that has been provided by $\mathcal{F}_{verify}$. Exactly these values are delivered by $\mathcal{S}$ to $\mathcal{F}_{vmpc}$, so the state of $\mathcal{F}_{vmpc}$ is consistent with $\mathcal{A}$.
   – *Step:* In the real world, for each $i \notin \mathcal{C}$, $\mathcal{A}$ chooses all the messages m^{ℓ}_{ji} for $j \in \mathcal{C}$ coming from i. By induction hypothesis, the rest of the messages m^{ℓ}_{ji} for $j \notin \mathcal{C}$ and the inputs/randomness $\boldsymbol{x}_i$,$\boldsymbol{r}_i$ of the

• **In the beginning**, $\mathcal{S}$ gets all the circuits $(C^{\ell}_{ij})^{n,n,r}_{i,j,\ell=1,1,1}$ from $\mathcal{F}_{vmpc}$. These are the same circuits that the parties would have obtained from $\mathcal{Z}$ in Π_{vmpc}.

1. *Initializing $\mathcal{F}_{verify}$:* $\mathcal{S}$ simulates the initialization of $\mathcal{F}_{verify}$.
2. *Randomness generation:* $\mathcal{S}$ simulates sending the messages $(\mathsf{commit\_rnd}, id)$ to $\mathcal{F}_{verify}$ for each input wire $id \leftarrow (i, \mathsf{r}, k)$. For all $i \in [n]$, the randomness $\boldsymbol{r}_i$ provided by $\mathcal{F}_{verify}$ is the same as the randomness $\boldsymbol{r}_i$ generated by $\mathcal{F}_{vmpc}$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, $\mathcal{S}$ simulates sending $(\mathsf{commit\_input}, \boldsymbol{x}_i, id)$ and $(\mathsf{commit\_input}, id)$ to $\mathcal{F}_{verify}$. For $i \in \mathcal{C}$, the value $\boldsymbol{x}^*_i$ is chosen by $\mathcal{A}$. $\mathcal{S}$ delivers this $\boldsymbol{x}^*_i$ to $\mathcal{F}_{vmpc}$.

• **For each round** $\ell \in [r]$, $\mathcal{S}$ needs to simulate computing the messages $\boldsymbol{m}^{\ell}_{ij} = C^{\ell}_{ij}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell-1}_{ni})$ for all $j \in \mathcal{C}$. If $i \in \mathcal{C}$, then the message $\boldsymbol{m}^{*\ell}_{ij}$ is generated by the adversary, and $\mathcal{S}$ delivers it to $\mathcal{F}_{vmpc}$. If $j \in \mathcal{C}$, then the message $\boldsymbol{m}^{\ell}_{ij}$ comes from $\mathcal{F}_{vmpc}$, and $\mathcal{S}$ delivers it to $\mathcal{A}$. In all cases, $\mathcal{S}$ simulates sending $(\mathsf{send\_msg}, m^{\ell}_{ijk}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for each entry m^{ℓ}_{ijk} of $\boldsymbol{m}^{\ell}_{ij}$.
• **After** r **rounds**, each (honest) party P_i should output $(\mathsf{output}, \boldsymbol{m}^r_{1i}, \ldots, \boldsymbol{m}^r_{ni})$ to $\mathcal{Z}$. This does not need to be simulated. Let $r' = r$ and $mlc_i[k] \leftarrow 0$ for all $k \in [n]$.
Alternatively, **at any time** before outputs are delivered to parties, if a message $(\mathsf{corrupt}, k)$ comes from $\mathcal{F}_{verify}$, $\mathcal{S}$ writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. In this case the outputs do not have not sent to $\mathcal{Z}$. $\mathcal{S}$ defines $\mathcal{B}_0 = \{k \mid (\mathsf{corrupt}, k) \text{ has been output}\}$, and sends $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$ to prevent it from outputting the results to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.
• **After** r' **rounds**:

1. $\mathcal{S}$ simulates sending messages $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n]$, $\ell \in [r']$, $k \in n^{\ell}_{ij}$.
2. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, $\mathcal{S}$ simulates sending (verify, id) to $\mathcal{F}_{verify}$. For each $k \in [n]$, $\mathcal{S}$ simulates the output bit b_k of $\mathcal{F}_{verify}$. If $k \in \mathcal{C}$, and $(f(id))(\boldsymbol{xid}) \neq \boldsymbol{yid}$ for the commitments of P_k in the inner state of $\mathcal{F}_{verify}$ maintained by $\mathcal{S}$, then $\mathcal{S}$ simulates $\mathcal{F}_{verify}$ outputting 0, and writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. Otherwise, simulates $\mathcal{F}_{verify}$ outputting 1, and writes $mlc_i[k] \leftarrow 0$. For all $k \notin \mathcal{C}$, it writes $mlc_i[k] \leftarrow 0$.

• **Finally**, $\mathcal{F}_{vmpc}$ outputs to each party P_i the set of parties $\mathcal{B}$ for which $\boldsymbol{m}^{*\ell}_{ij} \neq \boldsymbol{m}^{\ell}_{ij}$ has been provided by $\mathcal{S}$ at some point before. It now waits for a set of parties $\mathcal{B}_i$ from $\mathcal{S}$, containing the parties that will be additionally blamed by $\mathcal{B}_i$. Let $\mathcal{B}'_i = \{j \mid mlc_i[j] = 1\}$. $\mathcal{S}$ sends to $\mathcal{F}_{vmpc}$ the sets $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}'_i$, where $\mathcal{B}_0$ is the set defined in the execution phase.

Fig. 23: The simulator S_{vmpc} for verifiable computations

inner state of $\mathcal{F}_{vmpc}$ do not contradict with the view of $\mathcal{A}$. In Π_{vmpc}, $\mathcal{A}$ expects that an honest P_i will now compute each message $\boldsymbol{m}^{\ell+1} = C^{\ell}_{ij}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell}_{ni})$. In the inner state of $\mathcal{F}_{verify}$, the value $\boldsymbol{m}^{\ell+1}$ is computed in exactly the same way.

2. *The sets $\mathcal{B}_i$ of blamed parties.* $\mathcal{F}_{vmpc}$ computes all the messages $\boldsymbol{m}^{\ell}_{ij}$ and constructs the set $\mathcal{M}$ of parties j for whom $\boldsymbol{m}^{\ell}_{ij} \neq \boldsymbol{m}^{*\ell}_{ij}$, where $\boldsymbol{m}^{*\ell}_{ij}$ is the value provided by $\mathcal{S}$ (that was actually chosen by $\mathcal{A}$). After that, it receives a couple of messages $(\mathsf{blame}, i, \mathcal{B}_i)$ from $\mathcal{S}$, where $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}'_i$, and $\mathcal{B}_0 = \{k \mid (\mathsf{corrupt}, k)$ has come from $\mathcal{F}_{verify}$ in the execution phase $\}$. $\mathcal{F}_{vmpc}$ expects $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. First, we prove that $\mathcal{B}_i \subseteq \mathcal{C}$, i.e no honest party will be blamed.
   (a) For each $j \in \mathcal{B}_0$, a message $(\mathsf{corrupt}, j)$ has come from $\mathcal{F}_{verify}$ at some moment. By definition of $\mathcal{F}_{verify}$, no $(\mathsf{corrupt}, j)$ can be sent for $j \notin \mathcal{C}$. Hence $j \in \mathcal{C}$.
   (b) For each $j \in \mathcal{B}'_i$, the proof of P_j has not passed the final verification. For $j \notin \mathcal{C}$, S has committed to $\mathcal{F}_{verify}$ exactly those messages that correspond to the computation of f on the committed input, the randomness $\boldsymbol{r}_i$, and the incoming messages $\boldsymbol{m}^{\ell}_{ij}$. Hence $j \in \mathcal{C}$.

   Secondly, we prove that $\mathcal{M} \subseteq \mathcal{B}_i$, i.e all deviating parties will be blamed.
   (a) The first component of $\mathcal{M}$ is $\mathcal{B}_0$ for which $\mathcal{S}$ has sent $(\mathsf{stop}, \mathcal{B}_0)$ during the execution phase. The same set $\mathcal{B}_0$ is a component of each $\mathcal{B}_i$.
   (b) The second component $\mathcal{M}'$ of $\mathcal{M}$ are the parties P_i for whom inconsistency of $\boldsymbol{m}^{\ell}_{ij}$ happens in $\mathcal{F}_{vmpc}$. We show that if $i \notin \mathcal{B}_k$ for all $k \notin \mathcal{C}$, then $i \notin \mathcal{M}'$. Suppose by contrary that there is some $i \in \mathcal{M}'$, $i \notin \mathcal{B}_k$. If $i \notin \mathcal{B}_k$ for all $k \notin \mathcal{C}$, then the proof of i had succeeded for every C^{ℓ}_{ij}. For all $i, j \in [n]$,

Table 12. Costs of different phases of $\mathit{\Pi}_{vmpc}$ for one prover

phase	rounds	$\mathcal{F}_{transmit}$ op	#ops	# bits
pre	11	transmit	$n + nt$	$\mathsf{sh}_n \cdot (n\mu(3\log m + 1)(N_b \log m + N_r))$ $+\mathsf{sh}_n \cdot (n\kappa(3\log m + 1)(\log m + 1))$ $+\mathsf{sh}_n \cdot (nt \cdot M_r)$ $+cnt \cdot M_r$
		forward	nt	$\mathsf{sh}_n \cdot (nt \cdot M_r)$
		broadcast	$3 + 2t$	$\mathsf{sh}_n \cdot ((\mu - 1)\log m \cdot (N_b(4\log m + 1) + 4N_r))$ $+\mathsf{sh}_n \cdot (\kappa(3\log m + 1)(\log m + 1))$ $+2ct$
exec	$1 + r$	transmit	$n(1 + rn)$	$\mathsf{sh}_n \cdot (n \cdot M_x + M_c)$
post	2	transmit	n	$\mathsf{sh}_n \cdot (n \cdot M_c)$
		forward	n	$\mathsf{sh}_n \cdot (n \cdot M_c)$
	2	broadcast	1	$N_g(3\log m + 1)$
	2	broadcast	1	$\mathsf{sh}_n \cdot (n(N_g(3\log m + 1) + M_c))$

$\ell \in [r']$, i should have come up with the commitments $\boldsymbol{x}_i$, $\boldsymbol{r}_i$, $\boldsymbol{m}^{\ell}_{ij}$ such that $\mathcal{F}_{verify}$ outputs 1 on input (verify, id) for each output wire identifier id. By definition of $\mathcal{S}$, the committed $\boldsymbol{x}_i$ are chosen by $\mathcal{A}$ before the execution, in the input commitment phase, the randomness $\boldsymbol{r}_i$ is coming from the same distribution as the randomness generated by $\mathcal{F}_{vmpc}$, the incoming messages $\boldsymbol{m}^{\ell}_{ji}$ are those that are treated by $\mathcal{F}_{vmpc}$ as being sent to P_i by P_j, and the outgoing messages $\boldsymbol{m}^{\ell}_{ij}$ are the same that are computed by $\mathcal{F}_{vmpc}$ (the messages moving between two corrupted parties have been chosen by $\mathcal{A}$). Hence $\boldsymbol{m}^{\ell}_{ij} = C^{\ell}_{ij}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}^1_{1i}, \ldots, \boldsymbol{m}^{\ell-1}_{ni})$ for all $i, j \in [n]$, $\ell \in [r']$, so $i \notin \mathcal{M}'$. $\square$

Lemma 13. *Let $\mathit{\Pi}_{vmpc}$ use the implementation of $\mathit{\Pi}_{verify}$ that is built on top of $\mathit{\Pi}_{pre}$, $\mathit{\Pi}_{transmit}$, and $\mathit{\Pi}_{share}$. Let the initial protocol defined by the circuits C^{ℓ}_{ij} has the following parameters (for one prover):*

- *has r rounds;*
- *its largest ring is $\mathbb{Z}_m$;*
- *the number of transmitted bits of the protocol is M_c;*
- *the number of input and randomness bits is M_x and M_r respectively;*
- *the total number of used bit related gates (bit shift, bit decomposition, comparison, ring extension) is N_b;*
- *the number of multiplication and rotation gates (treating a rotation of length ℓ as ℓ gates) is N_r;*
- *the total number of input and output wires in the circuits (excluding the intermediate wires) is N_w;*

The resulting protocol may be seen as split into preprocessing, execution, and postprocessing phases, whose complexity upper bounds are given in Table 12 for the optimistic case (where the adversary does not attempt to cheat).

In the pessimistic case, up to the final verification, the number of rounds at most doubles, and the number of communicated bits increases at most $2n$ times. The cost of the final verification increases up to $\log(\max(N_w, N_b \log m + N_r))$ times.

Proof. Let $N_g := N_b + N_r$, $\mathsf{m} := \log m$. We have taken the numbers of communicated bits from the previously proved lemmas for $\mathit{\Pi}_{verify}$. In the optimistic case, $\mathcal{F}_{transmit}$ works in the cheap mode.

- The total cost of generating preprocessed tuples, taken from Lemma 7, is $t \cdot \mathsf{bc}_c + t \cdot \mathsf{bc}_c + n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot ((3\mathsf{m}+1)(\mu(N_b \cdot \mathsf{m} + N_r) + \kappa(\mathsf{m}+1)))} + \mathsf{bc}_{\mathsf{sh}_n \cdot (\kappa(3\mathsf{m}+1)(\mathsf{m}+1))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot \mathsf{m} + 2 \cdot N_r))} + \mathsf{bc}_{\mathsf{sh}_n \cdot ((\mu-1)\mathsf{m}(N_b \cdot (3\mathsf{m}+1) + 2 \cdot N_r))}$. Taking the number of rounds from Obs. 1, we get the total number of 11 rounds in the cheap mode of $\mathcal{F}_{transmit}$.
- The total cost of generating the randomness $nt \cdot (\mathsf{tr}_{M_r c} + \mathsf{tr}_{\mathsf{sh}_n \cdot M_r} + \mathsf{fwd}_{\mathsf{sh}_n \cdot M_r})$ is taken from Lemma 8. All the randomness of one prover can indeed be generated in parallel.
- Before the execution starts, each input has to be committed. The total cost of input commitment $n \cdot \mathsf{tr}_{\mathsf{sh}_n \cdot M_x}$ is taken from Lemma 8, where all the M_x bits of one prover are committed in parallel.

– The M_c bits of the original communication are transmitted in r rounds. On each round, up to $n(n-1)$ distinct transmissions may take place, since each of the n parties may send something to $n-1$ other parties. We upper bound it by rn^2 to get a nicer number.
– The remaining cost comes from the complexity of verify call of $\mathcal{F}_{verify}$. It consists of the following sequential blocks:
  - The total cost of mutually committing the messages $n \cdot (\mathsf{tr}_{\mathsf{sh}_n \cdot M_c} + \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c})$ is taken from Lemma 8, where all the messages can be committed in parallel.
  - The total number of broadcast bits $(N_b + N_r)(3\mathsf{m} + 1)$ of the postprocessing phase is taken from Lemma 9.
  - The total number of alleged zero bits $n \cdot \mathsf{sh}_n \cdot ((N_b + N_r)(3\mathsf{m} + 1) + M_c))$ is taken from Lemma 10. Here we assume that all the outputs of the circuits are exactly the communicated messages coming out of the circuits, so we do not introduce M_y.

In the pessimistic mode, $\mathcal{F}_{transmit}$ works in its expensive mode. As can be seen from Table 5, the number of rounds at most doubles, and the total communication increases up to $2n$ times. In the final opening, the function reveal of $\mathcal{F}_{transmit}$ is called instead of broadcast. Since we do not want to reveal all the messages that have been transmitted in parallel, the authentication paths of the Merkle tree for simultaneously sent values may need to be sent to each verifier, so that the signature may be checked. This gives a multiplicative overhead $\log M$ where M is the number of distinct elements sharing one signature. Since the total number of wires is N_w, we may assume that there cannot be more than N_w inputs, randomness, or communication elements committed in the same round. The maximum amount of distinct preprocessed tuples is $N_b \log m + N_r$. Hence the overhead can be at most $\log(\max(N_w, N_b \log m + N_r))$. □

C.6 Proof of the main theorem

We are now ready to prove Theorem 2. We take Π_{vmpc} that is build on top of Π_{verify}, which is in turn using Π_{share}, Π_{pre}, and $\Pi_{transmit}$.

Correctness For estimating the correctness error, we need to count the total number of messages sent using $\mathcal{F}_{transmit}$. By message we mean a bitstring that is signed with one signature. For this, we look at the Table 12.

– The tuple generation takes n parallel transmissions and 5 sequential broadcasts for one prover. Running this phase for all provers gives us $n(n+5)$ messages.
– The randomness generation takes $2nt$ transmissions and nt forwardings for each prover, which is $3n^2t$.
– In the beginning of the execution phase, each prover sends its $(n-1)$ input shares to the other parties. For all the n provers, this is $n(n-1)$.
– The execution phase has r rounds, where in each round each party may send something to the $n-1$ other parties, so it gives $rn(n-1)$ messages. In addition, each party sends the shares its input to $n-1$ other parties, which is $n(n-1)$ messages.
– After the execution phase, the prover broadcasts its public values. For all the provers, there are n broadcasts with a single signature.
– In the final check, a certain number of alleged zeroes is opened. For this, n shares should be broadcast by each prover, so there are n^2 messages.

The total number of sent messages is $n(n+5) + 3n^2t + n(n-1) + rn(n-1) + n(n-1) + n + n^2 \leq n^2(3n + r + 6)$ for $r \geq 1$,$n \geq 2$, which are reasonable assumptions for a multiparty protocol. By Lemma 3, the error of the underlying $\Pi_{transmit}$ is bounded by $n^2(3n + r + 6) \cdot \delta$. The other source of error is Π_{verify}. In order to achieve error at most 2^{η}, by Lemma 6 it is sufficient to take $\mu = 1 + \frac{\eta}{\mathsf{N_g}} \leq \eta$, and $\kappa = \max(\{(n^{1/\mu} + 1)\eta, n^{1/\mu} + \mu - 1\}) \leq \max(\{(2^{-\eta} + 1)\eta, 2^{-\eta} + \eta\}) \leq \eta{+}1$, which we will need when estimating the complexity of preprocessing phase.

Security We have proven that Π_{vmpc} securely implements $\mathcal{F}_{vmpc}$ in Lemma 12.

Complexity First, we estimate the complexity of the optimistic setting, where the adversary does not attempt to stop the protocol. We combine the numbers of Table 12 with the costs of particular $\mathcal{F}_{transmit}$

operations of Table. 5. Since the variables N_b, N_r, M_x, M_c, M_r are estimated for the entire computation of all the n parties, and the costs are linear w.r.t these values, we do not multiply each number by n to scale it to n provers. However, we still need to multiply the number of used $\mathcal{F}_{transmit}$ operations by n in the pre- and postprocessing phases. The only exception is the parameter κ of the preprocessing phase that is upper bounded by $\eta + 1$ for each separate proof, and which is not scaled to n parties (differently from μ). Hence we take everywhere $\kappa' := n\kappa$.

– *Preprocessing:* Transmit $\mathsf{sh}_n \cdot (n\mu(3\mathsf{m} + 1)(N_b\mathsf{m} + N_r) + n\kappa'(3\mathsf{m} + 1)(\mathsf{m} + 1) + nt \cdot M_r) + cntM_r$ bits. In order to achieve the reported correctness, we took $\mu \leq \eta$ and $\kappa \leq \eta + 1$ (so $\kappa' \leq n(\eta + 1)$). Since $3\mathsf{m} + 1 \leq 4\mathsf{m}$ and $(3\mathsf{m} + 1)(\mathsf{m} + 1) \leq 8\mathsf{m}^2$, and the total number of independent transmissions that need a signature is $(n + 2nt)$ for each prover, an upper bound of bit communication is

$$\mathsf{sh}_n \cdot (n\eta 4\mathsf{m}(N_b\mathsf{m} + N_r) + n^2(\eta + 1)8\mathsf{m}^2 + nt \cdot M_r) + cntM_r + n(n + 2nt)c' \ .$$

Similarly, the upper bound on $(\mu - 1)\mathsf{m} \cdot (N_b(4\mathsf{m} + 1) + 4N_r) + \kappa'(3\mathsf{m} + 1)(\mathsf{m} + 1) + 2ct$ is $\eta 4\mathsf{m}(N_b\mathsf{m} + N_r) + \eta 4\mathsf{m}N_b + n(\eta + 1)8\mathsf{m}^2 + 2cn$, so the upper bound on the broadcast bits is

$$\mathsf{sh}_n \cdot (n^2\eta 4\mathsf{m}(N_b\mathsf{m} + N_r) + n^2\eta\mathsf{m}N_b + n^3(\eta + 1)8\mathsf{m}^2) + 2cn^3 + 2n(1 + n)c' \ ,$$

since each of the $3+2t \leq 3+2n$ broadcasts of each prover needs a signature of c' bits, and each broadcast is up to n^2 times more expensive than a transmission.
Summing together the upper bounds, and putting all the non-leading terms into o, treating c and c' as constants, and assuming $n \leq N_g$ (each party computes at least one gate), we get the total number of bits upper bounded by

$$\mathsf{sh}_n \cdot (4n^2\eta\mathsf{m}(N_b\mathsf{m} + N_r) + n^2M_r + o(n\eta\mathsf{m}(N_b\mathsf{m} + N_r))) \ .$$

– *Execution:* all the n transmitted shares of n parties, and up to $n(n - 1)$ messages transmitted in each of the r rounds are signed, so the total number of signatures is $rn(n - 1) + n^2$. Treating c' as constant, we may write the total cost as $\mathsf{sh}_n \cdot (n \cdot M_x + M_c + o(rn^2))$.
– *Postprocessing:* Translating the values of Table 12 to actual communication gives us $(\mathsf{sh}_n \cdot (n \cdot M_c) + nc') + (\mathsf{sh}_n \cdot (n \cdot M_c) + 2nc') + n^2(N_g(3\mathsf{m} + 1) + nc') + n^2 \cdot n(\mathsf{sh}_n \cdot (N_g(3\mathsf{m} + 1)) + M_c + nc')$. Treating c' as constant, and assuming $n \leq N_g$, we may write it as $\mathsf{sh}_n \cdot (3n^2(n + 1)N_g\mathsf{m} + o(n^3(N_g + M_c)))$.

For estimating the numbers of the pessimistic setting, we look at Table. 5. The number of rounds for each expensive mode operation is twice as large as the same operation in the cheap mode, and the bit communication is up to $2n$ times larger. Another possibility for the adversary to increase the communication is to fail the last weak opening of alleged zeroes and force all the shares committed so far to be revealed. The weak opening may fail either if the prover clearly broadcasts inconsistent messages, or if some verifier complains that the broadcast values were not correct. In both cases, a strong opening pinpoints the party that has caused the weak opening to fail. Hence a covert adversary will not do it anyway. □

Discussion. For a fixed number of parties, we get the following complexities of different phases:

– *Preprocessing:* $O(\eta\mathsf{m}(N_b\mathsf{m} + N_r) + M_r)$.
– *Execution:* $O(M_x + M_c + r)$.
– *Postprocessing:* $O(N_g\mathsf{m} + M_c)$.

For $n = 5$, the constant of O is already quite large due to the exponential nature of share cost sh_n and the quadratic cost of broadcast. However, for $n = 3$, the constant is very small. The gates involving bit decomposition incur additional multiplicative overhead of $\mathsf{m} = \log m$, where m is the size of the ring in which the computation takes place. Otherwise, all the overheads are linear. Together with the special optimizations of Sec. 5.1, our verification method becomes very fast for 3-party protocols.