Peeter Laud*, Alisa Pankova, and Roman Jagomägis

Preprocessing Based Verification of Multiparty Protocols with Honest Majority

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. Our construction performs best with a small number of parties, and its main benefit is the total cost of the online and the offline 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.

Keywords: Secure multiparty computation, covert security

DOI 10.1515/popets-2017-0036
Received 2017-02-28; revised 2017-06-01; accepted 2017-06-02.

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 [33], 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 [9, 13, 19, 24], as well as various protocols optimized to solve concrete problems [12, 15, 18, 30].

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. [46]). 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. In this case, it suffices that, after execution, each party proves 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. Our construction performs best with a small number of parties, and its efficiency linearly depends on the total number of *local* non-linear operations performed by the parties. If the number of parties is $n > 3$, it also linearly depends on the number of bits communicated between the parties in the initial protocol, but this overhead is less significant.

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. Impor-

***Corresponding Author: Peeter Laud:** Cybernetica AS, E-mail: peeter.laud@cyber.ee
Alisa Pankova: Cybernetica AS, University of Tartu, STACC, E-mail: alisa.pankova@cyber.ee
Roman Jagomägis: Cybernetica AS, E-mail: roman.jagomagis@cyber.ee

tantly, 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.

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 [9] employed in the Sharemind platform [8], 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 [37], and its deployments [10, 34, 55] 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 passive adversary to a protocol secure against *covert* adversary [1] 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 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 [27], this contract can also specify appropriate punishments for being caught deviating from the protocol.

Moreover, the protocol set [9] is *private* against active adversaries, as long as no values are declassified [49]. 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 [42].

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. 5).

The reduction of the total cost of *actively* secure computation in the 3-party case, tolerating one corrupted party, 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 [38, 47], on additive sharing with MACs to check for correct behaviour [22, 24, 26], on Shamir's secret sharing [19, 53], and on the GMW protocol [33] paired with actively secure oblivious transfer [47]. 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 [24] or ABY [29]. 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 [29, 48, 50] or somewhat (fully) homomorphic encryption systems [11, 22] or oblivious transfer [47]. For ensuring the correctness of tuples, the generation is followed by a much cheaper correctness check [22]. 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, several identification mechanisms for SPDZ-like protocols have been proposed [4, 17, 54]. They scale better with the number of parties n than our protocol does. Let C be the circuit describing the functionality that the parties compute. In similar settings, our protocol would have complexity $O(|C|n^3)$. The complexity of the online phase of [4] is $O(|C|n^2)$, but their preprocessing phase needs $O(|C|n^3)$ ZK proofs. The complexity of [54] is $O(|C|n)$, but it is possible that different honest parties do not agree on the same set of cheaters, and enforcing this agreement requires $O(|C|n^2)$ cryptographic operations to repeat the computation of each party from its commitments. All honest parties agree on the same set of cheaters in [17], which has complexity $O(|C|n)$, but their method requires additional ZK proofs in the preprocessing phase, and the overhead of the online and the offline phases together is unclear. Our protocol uses the honest majority assumption to avoid cryptographic operations, and its benefit comes from the total cost of both the online and the offline phases.

For honest majority and three parties, a recent method [32] 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 [2, 20, 41]. We note that the general outline of our verification scheme is similar to [2] — 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 directly applied to computation over rings.

We apply our verification to the protocol set of Sharemind [8], 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 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 [34] 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 [45] 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 [28], 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 more efficient solution based on garbled circuits with TinyOT has emerged in parallel with our paper [35, 56]. Doing 1024 executions in parallel, it computes a single AES circuit in 6.7ms, which has the same order of magnitude as our solution. Their benchmarks are done using 10Gbps LAN while we have used 1Gbps LAN.

A protocol based on secret sharing over $\mathbb{Z}_2$ [47] would use the same circuit to perform integer multiplication. In [31], 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 [21], 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 [21]. In [22], 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 [19] (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 [25], 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 [28] 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 15µs. This is around two orders of magnitude faster than the performance reported above.

In concurrent work [36], the oblivious transfer methods of [31] 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 [23], amortized time 0.5µs 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 3.1ms, including the preprocessing.

3 Ideal Functionality

We propose a transformation that converts a passively secure multiparty protocol to one that is covertly secure under honest majority assumption. In this section, we formalize the initial protocol and state the properties that the transformed protocol should have.

Notation. Throughout this work, we use $\vec{x}$ to denote vectors, where x_i is the i-th coordinate of $\vec{x}$. Let $\vec{0} = (0, \dots, 0)$. All operations on vectors are defined elementwise. We denote $[n] = \{1, \dots, n\}$.

The UC framework. We specify our transformation in the universal composability (UC) framework [14]. In UC, the protocol runs in parallel with the adversary $\mathcal{A}$ that may corrupt the parties, getting access to their internal states, and the environment $\mathcal{Z}$ specifying the protocol inputs and receiving the outputs. There is also communication between $\mathcal{Z}$ and $\mathcal{A}$. The security proofs are based on indistinguishability of executing the real protocol Π from executing the ideal functionality $\mathcal{F}$ that precisely states the desired correctness and security properties. A protocol Π *UC-realizes* $\mathcal{F}$ if for any adversary $\mathcal{A}$ attacking Π there exists an adversary $\mathcal{A}_S$ attacking $\mathcal{F}$, such that no $\mathcal{Z}$ may distinguish whether it is interacting with $\mathcal{F}$ and $\mathcal{A}_S$, or with Π and $\mathcal{A}$.

The initial passively secure protocol. The protocol is run by n parties, indexed by $[n]$, where $\mathcal{C} \subseteq [n]$ are corrupted, $|\mathcal{C}| < n/2$. We denote $\mathcal{H} = [n] \backslash \mathcal{C}$. There is a secure channel between each pair of parties. The protocol is synchronous; it has r rounds, where the ℓ-th round computations of the party P_i, the results of which are sent to the party P_j, are given by a publicly known *arithmetic circuit* C^ℓ_{ij}. This circuit computes the ℓ-th round messages $\vec{m}^\ell_{ij}$ to the party $j \in [n]$ from the input $\vec{x}_i$, uniformly distributed randomness $\vec{r}_i$ and the messages $\vec{m}^k_{j'i}$ $(k < \ell)$ that P_i has received before. All values $\vec{x}_i, \vec{r}_i, \vec{m}^\ell_{ij}$ are vectors over rings $\mathbb{Z}_{2^N}$. The messages received during the r-th round comprise the *output of the protocol.*

Arithmetic circuits C^ℓ_{ij} over rings $\mathbb{Z}_{2^{n_1}}, \dots, \mathbb{Z}_{2^{n_K}}$ represent *local* computation of parties. Such a circuit consists of connected gates, performing arithmetic operations on inputs and producing outputs. An operation may be either an addition, a constant multiplication, or a multiplication in one of the rings $\mathbb{Z}_{2^{n_k}}$. It may also be "$x = \mathsf{trunc}(y)$" or "$y = \mathsf{zext}(x)$" for $x \in \mathbb{Z}_{2^n}$ and $y \in \mathbb{Z}_{2^m}$, where $n < m$. The first of them computes $x = y \bmod 2^n$, while the second lifts $x \in \mathbb{Z}_{2^n}$ to the larger ring $\mathbb{Z}_{2^m}$. Finally, there is an operation $(z_1, \ldots, z_m) = \mathsf{bd}(x)$ that decomposes $x \in \mathbb{Z}_{2^m}$ into bits $z_i \in \mathbb{Z}_2$. The gate outputs can be used as inputs of some other gates. The gate inputs that are not outputs of any other gates of C^{ℓ}_{ij} are called the *inputs* of C^{ℓ}_{ij}. Some gate outputs are treated as the final result of computing C^{ℓ}_{ij} on its inputs $\vec{x}$, and these are called the *outputs* of C^{ℓ}_{ij}. Computing the outputs $\vec{y}$ from the inputs $\vec{x}$ is denoted $\vec{y} = C^{\ell}_{ij}(\vec{x})$.

This set of gates is sufficient to represent any computation. Any other operations can be expressed as a composition of the available ones. Nevertheless, the verifications designed for special gates may be more efficient, and we discuss some of them in App. A.

Compared to the circuit C describing the functionality f that the parties compute, the sizes of C^{ℓ}_{ij} depend on the particular protocols used to compute f, and they may vary for different SMC platforms. In general, the total asymptotic size of local circuits of C taken altogether is $O(|C|n)$. For example, let C consist of linear gates, binary multiplication gates, and conversions between $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_2^m$. Let 3-party Sharemind protocols [9] be used. A single multiplication in $\mathbb{Z}_{2^m}$ needs 6 local multiplications in $\mathbb{Z}_{2^m}$. For bit decompositions and transitions between rings, we need to use share conversion algorithms, where transition from $\mathbb{Z}_{2^m}$ to $\mathbb{Z}_2^m$ has $64m$ AND gates and 64 bit decompositions in $\mathbb{Z}_{2^m}$, and transition from $\mathbb{Z}_2^m$ to $\mathbb{Z}_{2^m}$ has 64 multiplications and 96 bit decompositions in $\mathbb{Z}_{2^m}$. We have not counted local linear combinations since they do not affect verification cost.

The resulting covertly secure protocol. The protocol transformation is specified by the ideal functionality $\mathcal{F}_{vmpc}$ given in Fig. 1. Parties are given a set of publicly known arithmetic circuits C^{ℓ}_{ij} specifying the initial passively secure protocol. Honest parties use C^{ℓ}_{ij} to compute their outgoing messages m^{ℓ}_{ij}. The outgoing messages $m^{*\ell}_{ij}$ of corrupted parties are chosen by the adversary. After the computation ends, $\mathcal{F}_{vmpc}$ outputs to *all* honest parties a set $\mathcal{M}$ containing *all* corrupted parties P_i that have sent $\vec{m}^{*\ell}_{ij} \neq \vec{m}^{\ell}_{ij}$ to any honest party P_j during the execution, and also *all* parties that have caused the protocol to abort (the set $\mathcal{B}_0$). Even if only *some* rounds of the protocol are computed, all the parties that deviated from the protocol in completed rounds will be detected.

The sets $\mathcal{B}_i$ of parties that are finally blamed by P_i may contain some additional parties that do not belong to $\mathcal{M}$. This is related to unsuccessful cheating that may have been detected only by some parties. Since $\mathcal{B}_i \subseteq \mathcal{C}$, no honest parties (in $\mathcal{H}$) can be falsely blamed.

We note that if $\mathcal{M} = \emptyset$, then $\mathcal{A}_S$ does not learn anything that a semi-honest adversary could not learn. In this way, the transformed protocol defines a covertly secure execution of the protocol specified by C^{ℓ}_{ij}: even though the corrupted parties may cheat, they will be finally detected if they do it.

Differently from the initial passively secure protocol, the parties are no longer trusted to generate their randomness $\vec{r}_i$ themselves. Instead, $\vec{r}_i$ is generated by $\mathcal{F}_{vmpc}$, before the parties get their inputs $\vec{x}_i$ from $\mathcal{Z}$. At this point, the adversary may stop the functionality. This corresponds to the failure of randomness generation in the real protocol, and it is allowed by $\mathcal{F}_{vmpc}$, since it is safe to abort the computation that does not involve private inputs.

The aim of this paper is to construct a protocol transformation Π_{vmpc} such that, applying it to efficient passively secure protocols, we get new verifiable protocols that outperform state-of-the art protocols with similar security requirements. We prove Theorem 1, stating that Π_{vmpc} UC-realizes $\mathcal{F}_{vmpc}$.

Theorem 1. *Let $\mathcal{C}$ be the set of corrupted parties. If $|\mathcal{C}| < n/2$, then Π_{vmpc} UC-realizes $\mathcal{F}_{vmpc}$.*

The next section gives the construction of the protocol and sketches its security proof.

4 The Real Protocol

The initial passively secure protocol is defined by circuits C^{ℓ}_{ij} representing local computation of parties, as defined in Sec. 3. We transform such a protocol to a verifiable one, outlined as follows.

At the beginning of the **execution phase**, P_i commits itself to its inputs $\vec{x}_i$ and the randomness $\vec{r}_i$. The commitment method ensures that $\vec{r}_i$ is distributed uniformly. Then the parties start executing the protocol defined by C^{ℓ}_{ij}. During the execution, P_i computes the messages $\vec{m}^{\ell}_{ij}$ using C^{ℓ}_{ij}, committing itself and the receiver P_j to them. If P_i and P_j are both corrupted, then they are allowed to commit to arbitrary $\vec{m}^{*\ell}_{ij}$.

After the execution phase ends, the **verification phase** starts. Each party (the prover P) has to prove to the other parties (the verifiers $V_1, \ldots, V_{n-1}$) that it computed its local circuits C^{ℓ}_{ij} correctly w.r.t. commit-

• **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 it to $\mathcal{A}_S$.
For each $i \in [n]$, $\mathcal{F}_{vmpc}$ generates the randomness $\vec{r}_i$ for the party P_i. For $i \in \mathcal{C}$, it sends $(\mathsf{randomness}, i, \vec{r}_i)$ to $\mathcal{A}_S$. At this point, $\mathcal{A}_S$ **may stop** the functionality. If it continues, then for each $i \in \mathcal{H}$ [resp $i \in \mathcal{C}$], $\mathcal{F}_{vmpc}$ gets the inputs $(\mathsf{input}, \vec{x}_i)$ for the party P_i from $\mathcal{Z}$ [resp. $\mathcal{A}_S$].

• **For each round** $\ell \in [r]$, $i \in \mathcal{H}$ and $j \in [n]$, $\mathcal{F}_{vmpc}$ uses C_{ij}^{ℓ} to compute the message $\vec{m}_{ij}^{\ell}$ that the party P_i is supposed to deliver to P_j on the ℓ-th round. For all $j \in \mathcal{C}$, it sends $\vec{m}_{ij}^{\ell}$ to $\mathcal{A}_S$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\vec{m}_{ji}^{\ell}$ from $\mathcal{A}_S$.

• **After** r **rounds**, $\mathcal{F}_{vmpc}$ sends $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to each party P_i with $i \in \mathcal{H}$. Let $r' = r$ and $\mathcal{B}_0 = \emptyset$.
Alternatively, **at any time** before outputs are delivered to parties, $\mathcal{A}_S$ may send $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$, where $\mathcal{B}_0 \subseteq \mathcal{C}$ are the parties that cause the protocol to abort. In this case the outputs are not sent. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **After** r' **rounds**, $\mathcal{A}_S$ sends to $\mathcal{F}_{vmpc}$ the messages $\vec{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$.
$\mathcal{F}_{vmpc}$ defines the set of cheaters $\mathcal{M} = \mathcal{B}_0 \cup \{i \in \mathcal{C} \mid \exists j \in [n], \ell \in [r'] : \vec{m}_{ij}^{\ell} \neq C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})\}$.

• **Finally**, for each $i \in \mathcal{H}$, $\mathcal{A}_S$ sends $(\mathsf{blame}, i, \mathcal{B}_i)$ to $\mathcal{F}_{vmpc}$, with $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. $\mathcal{F}_{vmpc}$ forwards this message to P_i.

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

ted $\vec{x}_i$, $\vec{r}_i$, $\vec{m}_{ij}^{\ell}$, $\vec{m}_{j'i}^{\ell}$. All n interactive proofs of the n provers take place in parallel. It is possible that some verifiers V_i misbehave during the proof, and they should be blamed for that, even if they have not cheated in the execution phase. The proofs of all parties should terminate even if the corrupted verifiers leave the protocol.

The previous discussion is summarized by the ideal functionality $\mathcal{F}_{verify}$ given in Fig. 2. It treats all circuits C_{ij}^{ℓ} of one party P_i as a single circuit C_i. It assigns a unique index *id* to each input and output of the circuit. Such indexation makes it easier to see which commitments correspond to which inputs and outputs of C_i.

In the rest of this section, we describe the protocol implementing $\mathcal{F}_{verify}$, and the building blocks used by it. The protocol is going to have its own **preprocessing phase**, aiming to make the verification phase cheaper.

Throughout this section, we assume that the number of parties is 3, and at most one of them is corrupted. This assumption makes the presentation simpler, and it describes precisely our actual implementation, including all optimizations specific to 3 parties. We discuss in Sec. 6 how the transformation can be generalized to n parties with honest majority. In all protocols, we assume a fully malicious adversary and static corruptions.

4.1 Building Blocks

Ensuring Message Delivery. At any time during the protocol execution, a corrupted sender may refuse to send the message. If the receiver complains about not receiving it, the other parties do not know whether they should blame the sender or the receiver. It would be especially sad to allow a corrupted party to abort the verification phase in this way, so that the cheaters would not be pinpointed.

We want to achieve *identifiable abort*, i.e. if some party stops the protocol, it is blamed by all (honest) parties. For this purpose, we use the transmission functionality $\mathcal{F}_{transmit}$ proposed in [20] that we repeat in Figure 3. It allows to ensure message delivery, and to reveal previously received messages. The adversary may still interrupt transmission, but in this case a message $(\mathsf{cheater}, k)$ will be output to all honest parties, where $k \in \mathcal{C}$ has caused the interruption.

The outline of the protocol implementing $\mathcal{F}_{transmit}$ is the following. All transmitted messages are provided with signatures. This allows to reveal previously sent messages by letting the receiver present the sender's signature. If the receiver complains that the sender has not sent the message or did not sign it properly, then the sender is required to deliver the message to *each other party*, so that at least one other honest party forwards it to the receiver. If no honest party receives the message, the sender will be blamed by all of them. In a single adversary model (like UC), opening the message to all parties does not break privacy, since if there is a conflict between the sender and the receiver, then at least one of them is corrupted, and the adversary would get that message anyway. As long as all the parties follow the protocol, this opening will not be needed. In order to maintain synchronicity, the parties first wait for possible complaints, so formally $\mathcal{F}_{transmit}$ increases the number

$\mathcal{F}_{verify}$ uses arrays comm and sent for storing the commitments. It works with unique indices id, defining a commitment $\mathsf{comm}[id]$ and its ring size $m(id)$. The messages are first stored as $\mathsf{sent}[id]$ before being committed.

• **Initialization:** On input $(\mathsf{init}, (C_{ij}^{\ell})_{i,j,\ell}^{n,n,r})$ from all (honest) parties, where C_{ij}^{ℓ} is an arithmetic circuit, initialize comm and sent to empty arrays. For all $i \in [n]$, treat the composition of C_{ij}^{ℓ} for $j \in [n]$, $\ell \in [r]$ as a single circuit C_i. Generate a unique index xid_k^i for the k-th input of C_i, and yid_k^i for the k-th output of C_i. For all obtained indices id, read out from C_i the ring size $m(id)$ of the value indexed by id. Store C_i and all id, $m(id)$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, xid_k^i)$ from all (honest) parties, if $\mathsf{comm}[xid_k^i]$ is not defined yet, generate $r \xleftarrow{\$} \mathbb{Z}_{m(xid_k^i)}$, and assign $\mathsf{comm}[xid_k^i] \leftarrow r$. Output r to P_i. If $i \in \mathcal{C}$, output r also to $\mathcal{A}_S$.

• **Input Commitment:** On input $(\mathsf{commit\_input}, x, xid_k^i)$ from P_i, and $(\mathsf{commit\_input}, xid_k^i)$ from all (honest) parties, if $\mathsf{comm}[xid_k^i]$ is not defined yet, assign $\mathsf{comm}[xid_k^i] \leftarrow x$. If $i \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$.

• **Message Commitment:**

1. On input $(\mathsf{send\_msg}, x, yid_l^i, xid_k^j)$ from P_i, output x to P_j. If $i \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$. If $j \in \mathcal{C}$, output x to $\mathcal{A}_S$. If $\mathsf{sent}[yid_l^i]$ is not defined yet, assign $\mathsf{sent}[yid_l^i] \leftarrow x$.
2. On input $(\mathsf{commit\_msg}, yid_l^i, xid_k^j)$ from all (honest) parties, if $\mathsf{sent}[yid_l^i]$ is defined, and $\mathsf{comm}[yid_l^i]$ is not defined, assign $\mathsf{comm}[yid_l^i] = \mathsf{comm}[xid_k^j] \leftarrow \mathsf{sent}[yid_l^i]$. If $i, j \in \mathcal{C}$, assign $\mathsf{comm}[yid_l^i] = \mathsf{comm}[xid_k^j] \leftarrow x^*$, where x^* is chosen by $\mathcal{A}_S$.

• **Verification:** On input (verify, i) from all (honest) parties, if $\mathsf{comm}[id]$ have been defined for all identifiers id of C_i, construct vectors $\vec{x}$ and $\vec{y}$ such that $x_j \leftarrow \mathsf{comm}[xid_j^i]$, and $y_j \leftarrow \mathsf{comm}[yid_j^i]$. Compute $\vec{y'} \leftarrow C_i(\vec{x})$.
If $\vec{y'} - \vec{y} = \vec{0}$, output 1 to each party and $\mathcal{A}_S$. Otherwise, output 0 to each party and $\mathcal{A}_S$.

• **Cheater detection:** On input $(\mathsf{cheater}, k)$ from $\mathcal{A}_S$ for $k \in \mathcal{C}$, output $(\mathsf{cheater}, k)$ to each party. Do not accept any inputs from P_k anymore.

Fig. 2. The ideal functionality $\mathcal{F}_{verify}$ for verifying circuit computation w.r.t. the committed inputs and outputs

$\mathcal{F}_{transmit}$ works with unique message identifiers id, encoding a sender $s(id) \in [n]$ and a receiver $r(id) \in [n]$.

• **Initialization:** On input (init, s, r) from all (honest) parties, where s, r map a message identifier id to its sender and receiver respectively, deliver (init, s, r) to $\mathcal{A}_S$.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ from $P_{s(id)}$ and $(\mathsf{transmit}, id)$ from all (honest) parties, output (id, m) to $P_{r(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $r(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.

• **Reveal received message:** On input (reveal, id, i) from a party P_j that at any point received (id, m), output (id, m) to P_i. If $i \in \mathcal{C}$, output (id, m) also to $\mathcal{A}_S$. If both $s(id), j \in \mathcal{C}$, then $\mathcal{A}_S$ can ask $\mathcal{F}_{transmit}$ to output (id, m') for any m'.

• **Cheater detection:** If $\{s(id), r(id)\} \cap \mathcal{C} \neq \emptyset$, $\mathcal{A}_S$ may interrupt the transmission and ask $\mathcal{F}_{transmit}$ to output $(\mathsf{cheater}, k)$ to all parties for $k \in \mathcal{C} \cap \{s(id), r(id)\}$. If $(\mathsf{cheater}, k)$ is output for both $k \in \{s(id), r(id)\}$, then no (id, m) is output to the parties.

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

of rounds. In [20], cheap exception handling is proposed for this.

We use $\mathcal{F}_{transmit}$ as a blackbox. For simplicity, we write that a message has been transmitted or revealed using $\mathcal{F}_{transmit}$, and avoid using its formal interface. If $\mathcal{F}_{transmit}$ outputs a set of messages $(\mathsf{cheater}, k)$ for some round, the behaviour of honest parties depends on the protocol that uses $\mathcal{F}_{transmit}$. We describe this behaviour in the point "cheater detection" of the figures depicting our protocols, and it can be summarized as follows. In the preprocessing phase, each party aborts. In the execution phase, each party includes k into the set $\mathcal{M}$ of cheaters and jumps to the verification phase. In the verification phase, each party includes k into the set $\mathcal{M}$ of cheaters, and the verification proceeds with the remaining parties, which is always possible due to an honest majority and the nature of commitments (if $n = 3$, the verification stops and the remaining parties are claimed honest, since the cheater has already been detected).

Broadcast and opening. For 3 parties, broadcast with identifiable abort can be built on top of $\mathcal{F}_{transmit}$. If a party P wants to broadcast a message

m, it uses $\mathcal{F}_{transmit}$ to deliver m to the other receiver. Upon receiving m_i and m_j respectively, the parties P_i and P_j exchange $h_i = H(m_i)$ and $h_j = H(m_j)$, where H is a collision-resistant hash function, i.e. $h_i = h_j$ implies $m_i = m_j$ with high probability, even if the adversary chooses m_i and m_j. If $h_i \neq h_j$, then P_i and P_j reveal m_i and m_j to each other using $\mathcal{F}_{transmit}$. If it turns out that P_i has cheated, then P_j may proceed with m_j. If P has cheated, then P_i and P_j have agreed on cheater's identity, and they behave in the same way as if cheating was detected using $\mathcal{F}_{transmit}$.

To open a previously transmitted message to both other parties, the hash exchange is not necessary. Since at most one party of 3 is corrupted, $\mathcal{F}_{transmit}$ does not allow two different messages (id, m) and (id, m') to be revealed to different parties.

Throughout this section, by *broadcast* and by *opening* we mean these $\mathcal{F}_{transmit}$-based protocols. In order to avoid ambiguity, no other definitions of broadcast and opening are used.

Commitments. All the inputs, the randomness, and the messages of the prover P are committed by additively sharing them among the verifiers V_1 and V_2. To commit $x \in \mathbb{Z}_m$, the prover P generates random $x^1 \overset{\$}{\leftarrow} \mathbb{Z}_m$ and computes $x^2 = x - x^1$ in $\mathbb{Z}_m$. P uses $\mathcal{F}_{transmit}$ to deliver x^i to V_i. Using $\mathcal{F}_{transmit}$ allows to argue about the authenticity of x^i later, if there are conflicts between P and V_i. We write $[\![x]\!]$ to denote the sharing of x, and $x = x^1 + x^2$ to denote that x was shared to the particular shares x^1 and x^2. The resulting commitment scheme is homomorphic.

Throughout this section, by *commitment* we mean this sharing-based commitment. In order to avoid ambiguity, no other definition of commitment is used.

Precomputed tuples. To reduce the work of the verifiers, we add a preprocessing phase generating correlated randomness, i.e. *precomputed tuples*. They are secret-shared among the verifiers, who have been convinced that the correlation holds. The prover P gets all the shares. The *verified multiplication triples* are triples (a, b, c) from some ring, such that $a \cdot b = c$. The *trusted bits* are values b from some ring $\mathbb{Z}_m$, $m > 2$, such that $b \in \{0, 1\}$. The tuple generation may fail, and it is possible that the deviator cannot be identified. This is not a problem since no private data is involved into this phase. However, it should be possible to open later the shares of precomputed tuples on demand, and if the tuples have already been generated once, the opening should always succeed. We formalize the functionality $\mathcal{F}_{pre}$ in Fig. 4, and we give the implementation of $\mathcal{F}_{pre}$ in Sec. 4.2.

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

The protocol Π_{pre} implementing $\mathcal{F}_{pre}$ is given in Fig. 5. The prover P, allowed to know the sharings, generates and shares the bits and the triples itself. The shares are delivered to the verifiers through $\mathcal{F}_{transmit}$, so that the shares could be opened later, and P gets committed to the values it has generated. The prover is interested in generating the tuples randomly, because his (and only his) privacy depends on it.

The verifiers check whether P generated the tuples correctly. The check is based on cut-and-choose and pairwise check, similarly to e.g. [24, 32]. The check is probabilistic, and it depends on parameters μ and κ. In particular, in order to obtain u tuples of certain kind, $\mu \cdot u + \kappa$ tuples have to be generated and shared by P.

First, the parties agree on a joint random seed, defining a random permutation π of the tuples. In the cut-and-choose step, they take the first κ randomly permuted tuples and open them. The check fails if any of the opened tuples are wrong. If all of them are correct, then only a small fraction of remaining tuples is wrong.

The remaining tuples are partitioned into groups of size μ. In each group, the first $\mu - 1$ tuples are used to verify the μ-th one in $\mu - 1$ pairwise checks. The core of each check is using homomorphic properties of secret sharing to compute a certain linear combination z of the tuple elements and verify that $z = 0$ (we call such z an *alleged zero*). The check is certain to fail if only one of the tuples in the pair is correct. Let $[\![z]\!]$ be computed as in Fig. 5. We show that, if $z = 0$, and one tuple is correct, then the other tuple is certainly also correct.

Trusted bits. Let the bit $[\![b']\!]$ in a ring $\mathbb{Z}_m$ be used to verify that $[\![b]\!]$ is a bit. Let $b' \in \{0, 1\}$. The prover broadcasts a bit c indicating whether $b = b'$.

- If $c = 1$, then $[\![z]\!] = [\![b]\!] - [\![b']\!]$ is computed. If $z = 0$, then it should be $b = b' \in \{0, 1\}$.
- If $c = 0$, then $[\![z]\!] = [\![b]\!] + [\![b']\!] - 1$ is computed. If $z = 0$, then it should be $b = 1 - b' \in \{0, 1\}$.
- If $c \notin \{0, 1\}$, the protocol aborts.

Multiplication triples. Let the triple $([\![a']\!], [\![b']\!], [\![c']\!])$ be used to verify the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. Let $c' = a' \cdot b'$. The values $\hat{a} = a - a'$, $\hat{b} = b - b'$ are computed and declassified by the verifiers, so there is no way for P to cheat with them. The verifiers compute and declassify $[\![z]\!] = \hat{a} \cdot [\![b]\!] + \hat{b} \cdot [\![a']\!] + [\![c']\!] - [\![c]\!]$. Since $c' = a' \cdot b'$, we have $z = \hat{a} \cdot b + \hat{b} \cdot a' + a' \cdot b' - c = a \cdot b - c$. Therefore, if $z = 0$, then $a \cdot b = c$.

One instance of $\mathcal{F}_{pre}$ is used to generate u preprocessed tuples in a ring $\mathbb{Z}_m$ for one party P_i. It works with unique indices id defining a multiplication triple $\mathsf{triple}[id]$ or a trusted bit $\mathsf{bit}[id]$.

• **Initialization:** On input (init, i, m, u) from all (honest) parties, where $i \in [n]$ is a party index, initialize triple and bit to empty arrays. Generate u unique indices id. Store m, u, i and all id for further use. As shorthand notation, let $P = P_i$. Let V_1 and V_2 denote the verifiers of P.

• **Trusted bit generation:** On input (bit, id) from all (honest) parties, check if $\mathsf{bit}[id]$ exists. If it does, take $(\vec{b}^1, \vec{b}^2) \leftarrow \mathsf{bit}[id]$. Otherwise, generate a vector of random bits $\vec{b} \xleftarrow{\$} \mathbb{Z}_2^u$. If $i \in \mathcal{C}$, then $\vec{b} \in \mathbb{Z}_2^u$ is chosen by $\mathcal{A}_S$. Share elementwise $\vec{b} = \vec{b}^1 + \vec{b}^2$ over $\mathbb{Z}_m$. Assign $\mathsf{bit}[id] \leftarrow (\vec{b}^1, \vec{b}^2)$.
Output $\vec{b}^j$ to V_j. Output $(\vec{b}^1, \vec{b}^2)$ to P. For $k \in \mathcal{C}$, send $\vec{b}^k$ to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send $(\vec{b}^1, \vec{b}^2)$ to $\mathcal{A}_S$.

• **Multiplication triple generation:** On input (triple, id) from all (honest) parties, check if $\mathsf{triple}[id]$ exists. If it does, take $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2)) \leftarrow \mathsf{triple}[id]$. Otherwise, generate random vectors $\vec{a} \xleftarrow{\$} \mathbb{Z}_m^u$, $\vec{b} \xleftarrow{\$} \mathbb{Z}_m^u$, and compute elementwise $\vec{c} \leftarrow \vec{a} \cdot \vec{b}$. If $i \in \mathcal{C}$, then $\vec{a}, \vec{b} \in \mathbb{Z}_m^u$ are chosen by $\mathcal{A}_S$. Share elementwise $\vec{a} = \vec{a}^1 + \vec{a}^2$, $\vec{b} = \vec{b}^1 + \vec{b}^2$, and $\vec{c} = \vec{c}^1 + \vec{c}^2$ over $\mathbb{Z}_m$. Assign $\mathsf{triple}[id] \leftarrow ((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$.

Output $(\vec{a}^j, \vec{b}^j, \vec{c}^j)$ to V_j. Output $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$ to P. For $k \in \mathcal{C}$, send $(\vec{a}^k, \vec{b}^k, \vec{c}^k)$ also to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$ to $\mathcal{A}_S$.

• **Share open:** On input $(\mathsf{share\_open}, id, k)$ from all (honest) parties, if the bit b [a triple (a, b, c)] identified by id has already been generated, output the share b^k [the shares (a^k, b^k, c^k)] to each party and to $\mathcal{A}_S$.

• **Stopping:** At any time, on input (stop) from $\mathcal{A}_S$, stop the functionality and output $\perp$ to all parties.

Fig. 4. Ideal functionality $\mathcal{F}_{pre}$

The bit c denoting whether $b = b'$ and the values $\hat{a} = a - a'$, $\hat{b} = b - b'$ are all distributed uniformly in the corresponding rings, since one of the tuples serves as a mask for the other tuple. All these values can be simulated in the security proof, being sampled uniformly.

In the protocol of Fig. 5, the verifiers do not check $\vec{z} = \vec{0}$ directly. Instead, they exchange $h_1 = H(\vec{z}^1)$ and $h_2 = H(-\vec{z}^2)$, where H is a collision-resistant hash function, and $\vec{z}^i$ is the share of $\vec{z}$ held by V_i. Similarly to the broadcast of Sec. 4.1, if $h_1 = h_2$, it should with high probability be $\vec{z}^1 = -\vec{z}^2$, implying $\vec{z} = \vec{z}^1 + \vec{z}^2 = \vec{0}$. These shares are easy to simulate in the security proof since one share $\vec{z}^i$ is already known to the adversary, and the other one can be computed as $\vec{z}^j = -\vec{z}^i$. The hash h_i can be computed directly from $\vec{z}^i$.

A corrupted verifier may intentionally provide wrong h_i, $\hat{a}^i$, or $\hat{b}^i$, causing the correctness check to fail. It will not be clear whether P or V_i is guilty. Such failure is allowed by $\mathcal{F}_{pre}$ since it does not handle private data. Alternatively, all shares could be revealed through $\mathcal{F}_{transmit}$ to identify the cheater.

If all $\mu - 1$ checks succeed, then the first $\mu - 1$ tuples in each group are discarded and only the last one is used. Since a pairwise check passes only if *both* tuples are incorrect, the corrupted prover needs to make *all* μ tuples in a group incorrect to make a single incorrect tuple accepted, and this probability is made negligible by adjusting the parameters μ and κ.

A combinatorial analysis, omitted due to space constraints (its results are quite similar to [32]), shows that values μ and κ do not need to be large to bound the prover's cheating probability by 2^{-80}. For example, if $u = 2^{20}$, then it is sufficient to take $\mu = 5$ and $\kappa = 1300$. If $u = 2^{30}$ then $\mu = 4$ and $\kappa = 14500$ are sufficient. At the other extreme, if $u = 10$, then $\mu = 26$ and $\kappa = 168$ are sufficient for the same security level.

In a finite field, more efficient methods than cut-and-choose and pairwise check are available. For example, we can replace them with an application of linear error correcting codes [3]. This technique allows to construct u verified tuples from only $u+\kappa$ initial ones, where κ is proportional to the security parameter η.

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

Lemma 1. *Let $n = 3$. Assuming at most one corrupted party, the protocol Π_{pre} UC-realizes $\mathcal{F}_{pre}$ in $\mathcal{F}_{transmit}$-hybrid model.*

The cut-and-choose with pairwise check works similarly to [24, 32], so we refer to [32] for a more formal proof.

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

The protocol Π_{verify} implementing $\mathcal{F}_{verify}$ is given in Fig. 6-7. All communication between parties takes place

• **Initialization:** The protocol starts with each party getting the input (init, i, m, u), where P_i is the prover, m is the ring size, and u is the number of tuples to be generated. The protocol uses parameters μ and κ that depend on the security parameter. As shorthand notation, let $P = P_i$. Let V_1 and V_2 denote the verifiers of P.

• **Trusted bits:** On input (bit, id), the party expects that the following protocol is executed:

1. The party P generates $(\mu \cdot u + \kappa)$ random bits $b \xleftarrow{\$} \mathbb{Z}_2$.
2. P shares $b = b^1 + b^2$ in $\mathbb{Z}_m$, and sends b^i to V_i using $\mathcal{F}_{transmit}$.
3. The parties agree on a public random permutation π of generated bits $\vec{b}$. For $b \in \{b_{\pi(1)}, \ldots, b_{\pi(\kappa)}\}$, V_1 and V_2 reveal b^1 and b^2 through $\mathcal{F}_{transmit}$, and each party computes $b = b^1 + b^2$. If $b \notin \{0, 1\}$, each party outputs $\perp$.
4. The remaining bits are split into groups of μ, where the first $\mu - 1$ bits are used to verify the μ-th one. Let the bit $[\![b']\!]$ be used to verify that $[\![b]\!]$ is a bit. P broadcasts a bit c indicating whether $b = b'$ or not. If $c = 1$ (indicating $b = b'$), the verifiers compute $[\![z]\!] = [\![b]\!] - [\![b']\!]$. If $c = 1$, the verifiers compute $[\![z]\!] = [\![b]\!] + [\![b']\!] - 1$. This process is repeated $\mu - 1$ times with the same $[\![b]\!]$, choosing different $[\![b']\!]$.
5. After V_1 and V_2 have computed $[\![z]\!]$ for all bit pairs, they are holding the vector shares $\vec{z}^1$ and $\vec{z}^2$ respectively. They compute and exchange hashes $h_1 = H(\vec{z}^1)$ and $h_2 = H(-\vec{z}^2)$, checking if $h_1 = h_2$. If the check fails, V_1 and V_2 inform P about the failure, and each party outputs $\perp$. If it succeeds, V_1 and V_2 inform P about success. For each of the remaining u bits b, P outputs b, and V_1 and V_2 output the shares b^1 and b^2 respectively.

• **Multiplication triples:** On input (triple, id), the party expects that the following protocol is executed:

1. The party P generates $(\mu \cdot u + \kappa)$ triples (a, b, c) such that $a \xleftarrow{\$} \mathbb{Z}_m$, $b \xleftarrow{\$} \mathbb{Z}_m$, $c = a \cdot b$.
2. P shares $a = a^1 + a^2$, $b = b^1 + b^2$, $c = c^1 + c^2$ in $\mathbb{Z}_m$, and sends (a^i, b^i, c^i) to V_i using $\mathcal{F}_{transmit}$.
3. The parties agree on a public random permutation π of generated triples. For $a \in \{a_{\pi(1)}, \ldots, a_{\pi(\kappa)}\}$, $b \in \{b_{\pi(1)}, \ldots, b_{\pi(\kappa)}\}$, $c \in \{c_{\pi(1)}, \ldots, c_{\pi(\kappa)}\}$, V_1 and V_2 reveal a^1, b^1, c^1, a^2, b^2, c^2 through $\mathcal{F}_{transmit}$. Each party computes $a = a^1 + a^2$, $b = b^1 + b^2$, $c = c^1 + c^2$. If $a \cdot b \neq c$, each party outputs $\perp$.
4. The remaining triples are split into groups of μ, where the first $\mu - 1$ triples are used to verify the μ-th one. Let the triple $([\![a']\!], [\![b']\!], [\![c']\!])$ be used to verify the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. The verifiers compute $[\![\hat{a}]\!] = [\![a]\!] - [\![a']\!]$ and $[\![\hat{b}]\!] = [\![b]\!] - [\![b']\!]$, and declassify $\hat{a}, \hat{b}$ by exchanging the shares $\hat{a}^i = a^i - a'^i$ and $\hat{b}^i = b^i - b'^i$. Then they compute $[\![z]\!] = \hat{a} \cdot [\![b]\!] + \hat{b} \cdot [\![a']\!] + [\![c']\!] - [\![c]\!]$. This process is repeated $\mu - 1$ times with the same triple $([\![a]\!], [\![b]\!], [\![c]\!])$, choosing different triples $([\![a]\!]', [\![b]\!]', [\![c]\!]')$.
5. The checks $z = 0$ are done similarly to the step (5) of trusted bits. If the check fails, each party outputs $\perp$. If it succeeds, for each of the remaining u triples (a, b, c), P outputs (a, b, c), and V_1 and V_2 output the shares (a^1, b^1, c^1) and (a^2, b^2, c^2) respectively.

• **Share open:** On input $(\mathsf{share\_open}, id, k)$, the party P_k opens its share of the tuple identified by id using $\mathcal{F}_{transmit}$. If P_k refuses to open, each party outputs $(\mathsf{cheater}, k)$.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{transmit}$, each party outputs $\perp$.

Fig. 5. Real protocol Π_{pre}

using $\mathcal{F}_{transmit}$. In this way, if a party refuses to send a properly formatted message, it will be publicly blamed. If the prover is blamed, then its proof does not proceed further. If one of the verifiers is blamed, then the proofs of other parties may be halted, since they should be honest assuming at most one corrupted party. Hence, without loss of generality, we assume that all the transmissions of $\mathcal{F}_{transmit}$ succeed.

Initialization. The initialization fixes the circuits C_{ij}^{ℓ} that are going to be verified. A sufficient number of precomputed tuples is generated by $\mathcal{F}_{pre}$. The number of these tuples and their types depends on the gates of C_{ij}^{ℓ}, described more precisely in Fig. 6. The verification phase clarifies why exactly these tuples are generated.

Randomness commitment. The prover P must fairly choose the (uniformly distributed) randomness it is going to use as the input of the composition of its circuits C_i, and commit to it. For this purpose, the *verifiers* jointly generate this randomness. Each verifier V_j generates a uniformly distributed r_j and uses $\mathcal{F}_{transmit}$ to deliver r_j to P. After receiving r_1 and r_2, P takes the randomness $r = r_1 + r_2$ that is additively shared among V_1 and V_2. Since at least one verifier V_j is honest, and the other verifier does not know anything about the value r_j, the randomness r is distributed uniformly.

• **Initialization:** The protocol starts with each party getting the input $(\mathsf{init}, (C^{\ell}_{ij})^{n,n,r}_{i,j,\ell})$, where the composition of C^{ℓ}_{ij} for each i is denoted C_i. The circuit defines the ring sizes $m(xid^i_k)$ of inputs and $m(yid^i_k)$ of outputs of C_i. As a shorthand notation, let $P = P_i$, $P' = P_j$. Let V_1, V_2 be the verifiers of P, and V'_1, V'_2 the verifiers of P'. The subroutine $\mathcal{F}_{pre}$ is called to generate a sufficient number of precomputed tuples for each party. The number of tuples and their types depend on the gates of the circuits C_i.

1. *Linear combination, conversion to a smaller ring:* no tuples needed;
2. *Multiplication in* $\mathbb{Z}_m$*:* one multiplication triple over $\mathbb{Z}_m$;
3. *Bit decomposition in* $\mathbb{Z}_{2^m}$*:* m trusted bits over $\mathbb{Z}_{2^m}$;
4. *Conversion from* $\mathbb{Z}_{2^n}$ *to a larger ring* $\mathbb{Z}_{2^m}$*:* n trusted bits over $\mathbb{Z}_{2^m}$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, xid^i_k)$, V_1 generates $r_1 \xleftarrow{\$} \mathbb{Z}_m$, and V_2 generates $r_2 \xleftarrow{\$} \mathbb{Z}_m$. They send r_1 and r_2 to P using $\mathcal{F}_{transmit}$. On input $(\mathsf{commit\_rnd}, xid^i_k)$, P expects to receive r_1 and r_2 from $\mathcal{F}_{transmit}$. It takes $r = r_1 + r_2$. Now r is treated as the committed k-th input of C_i.

• **Input Commitment:** On input $(\mathsf{commit\_input}, x, xid^i_k)$, P shares $x = x^1 + x^2$ in $\mathbb{Z}_m$ and uses $\mathcal{F}_{transmit}$ to deliver x^1 to V_1 and x^2 to V_2. On input $(\mathsf{commit\_input}, xid^i_k)$, V_1 and V_2 expect to receive x^1 and x^2 respectively from $\mathcal{F}_{transmit}$. Now x is treated as the committed k-th input of C_i.

• **Message Commitment:**
1. On input $(\mathsf{send\_msg}, x, yid^i_l, xid^j_k)$, P uses $\mathcal{F}_{transmit}$ to deliver x to P'. On input $(\mathsf{send\_msg}, yid^i_l, xid^j_k)$, P' expects to receive x from $\mathcal{F}_{transmit}$.
2. On input $(\mathsf{commit\_msg}, yid^i_l, xid^j_k)$, the verifier $V_1 = P'$ takes the share $m^1 = m$, and the other verifier $V_2 \neq P'$ takes the share $m^2 = 0$. Analogously, treating P' as a prover, the verifier $V'_1 = P$ takes the share $m^1 = m$, and the other verifier $V'_2 \neq P$ takes the share $m^2 = 0$. Now m is treated as the committed l-th output of C_i and the committed k-th input of C_j.

Fig. 6. Real protocol Π_{verify} (initialization and commitments)

In the security proof, the simulator is able to simulate exactly the same r that has been chosen by $\mathcal{F}_{verify}$, taking $r_j = r - r_i$ after the adversary has chosen r_i for the corrupted verifier.

Input commitment. At the beginning of protocol execution, P commits to its input x by sharing it as $x = x_1 + x_2$ and using $\mathcal{F}_{transmit}$ to deliver x_i to V_i. The share issued to the corrupted verifier is distributed uniformly and is easy to simulate. A corrupted prover may share any x in an arbitrary way, as allowed by $\mathcal{F}_{verify}$.

Message commitment. During the protocol execution, the sender transmits each message m using $\mathcal{F}_{transmit}$. The sender can be the prover P as well as some other party P'. As the result, each message m that has been sent or received by P is known at least to one verifier V_1 or V_2 that has been on the other side of the communication. Since each such message m has been delivered using $\mathcal{F}_{transmit}$, it is possible to prove its authenticity later, and hence both the sender and the receiver have been committed to the same m. For both of them, it can be viewed as being additively shared among the verifiers as $m = m + 0$.

Verifying local computations. The local computation of P_i is represented by circuits C^{ℓ}_{ij} turning already received messages to new messages of the next round (see Sec. 3).

The circuits are verified gate-by-gate. For each gate, we have the following setup. The gate operation *op* takes k inputs in some ring $\mathbb{Z}_m$ and produces l outputs in some ring $\mathbb{Z}_{m'}$. The input values are shared as $[\![x_1]\!], \ldots, [\![x_k]\!]$ among the verifiers. The prover knows all these shares. During the computation of the circuit, 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 $[\![y_1]\!], \ldots, [\![y_l]\!]$ shared among the verifiers, where the prover again knows the shares of both verifiers, but no verifier has learned anything new. The verification step also gives us a number of *alleged zeroes* — shared values $[\![z_1]\!], \ldots, [\![z_s]\!]$ (all known to the prover), such that, if $z_1 = \cdots = z_s = 0$ then the verifiers are sure that the sharings $[\![y_1]\!], \ldots, [\![y_l]\!]$ indeed correspond to $y_1, \ldots, y_l$. All these equality checks are deferred to be succinctly verified one round later.

Repeating this process gate by gate, the verifiers finally obtain a sharing $[\![y]\!]$ of some output of the circuit from the commitments to its inputs. The prover has

• **Verification (1st round):** On input (verify, i), the prover P broadcasts some hints that will be used by V_1 and V_2 to localize their computation. These values depend on the gates of the circuit C_i.

1. *Linear combination.* No broadcasts needed.
2. *Conversion from $\mathbb{Z}_{2^n}$ to a smaller ring $\mathbb{Z}_{2^m}$.* No broadcasts needed.
3. *Multiplication in $\mathbb{Z}_m$.* Let $[\![y]\!] = [\![x_1]\!] \cdot [\![x_2]\!]$ be the statement being verified. Let $([\![a]\!], [\![b]\!], [\![c]\!])$ be a precomputed multiplication triple over $\mathbb{Z}_m$. P broadcasts $\hat{x}_1 = x_1 - a$ and $\hat{x}_2 = x_2 - b$.
4. *Bit decomposition in $\mathbb{Z}_{2^m}$.* Let $([\![y_0]\!], \dots, [\![y_{m-1}]\!]) = \mathsf{bd}([\![x]\!])$ be the statement being verified. Let $[\![b_0]\!], \dots, [\![b_{m-1}]\!]$ be precomputed trusted bits, shared over $\mathbb{Z}_m$. P broadcasts bits $c_0, \dots, c_{m-1}$, where $c_k = 1$ iff $b_k = y_k$.
5. *Conversion from $\mathbb{Z}_{2^n}$ to a larger ring $\mathbb{Z}_{2^m}$.* Let $y = \mathsf{zext}(x)$ be the statement being verified. Let $[\![b_0]\!], \dots, [\![b_{n-1}]\!]$ be precomputed trusted bits, shared over $\mathbb{Z}_{2^m}$. P performs bit decomposition of x over $\mathbb{Z}_{2^m}$, getting m bits x_k. It takes the first n of these bits, and broadcasts $c_0, \dots, c_{n-1}$, where $c_k = 1$ iff $b_k = x_k$.

• **Verification (2nd round):** After the broadcasts have been done, the verifiers start computing C_i locally on shares, collecting the alleged zeroes. V_1 and V_2 compute the gates as follows.

1. *Linear combination.* Let $y = \sum_{j=1}^{t} c_j \cdot x_j$ be the statement being verified. Compute $[\![y]\!] = \sum_{j=1}^{t} c_j \cdot [\![x_j]\!]$.
2. *Conversion from $\mathbb{Z}_{2^n}$ to a smaller ring $\mathbb{Z}_{2^m}$.* Drop $n - m$ highest bits from all shares of x.
3. *Multiplication in $\mathbb{Z}_m$.* Using $\hat{x}_1$ and $\hat{x}_2$ that P has broadcast before, compute $[\![y]\!] = \hat{x_1} \cdot [\![b]\!] + \hat{x_2} \cdot [\![a]\!] + [\![c]\!] + \hat{x_1} \cdot \hat{x_2}$. Compute the alleged zeroes $[\![z_1]\!] = [\![x_1]\!] - [\![a]\!] - \hat{x}_1$ and $[\![z_2]\!] = [\![x_2]\!] - [\![b]\!] - \hat{x}_2$.
4. *Bit decomposition in $\mathbb{Z}_{2^m}$.* Using the bits c_k that P has broadcast before, take $[\![y_k]\!] = [\![b_k]\!]$ if $c_k = 0$, and $[\![y_k]\!] = 1 - [\![b_k]\!]$, if $c_k = 1$. Compute the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=0}^{m-1} 2^k \cdot [\![y_k]\!]$.
5. *Conversion from $\mathbb{Z}_{2^n}$ to a larger ring $\mathbb{Z}_{2^m}$*: Using the broadcast bits $c_0, \dots, c_{n-1}$, perform the bit decomposition of $[\![x]\!]$, obtaining the shared bits $[\![y_0]\!], \dots, [\![y_{n-1}]\!]$; the bits are shared over the ring $\mathbb{Z}_{2^m}$. Compute $[\![y]\!] = \sum_{i=0}^{n-1} 2^k \cdot [\![y_k]\!]$ and the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=1}^{n-1} 2^k \cdot \mathsf{trunc}([\![y_k]\!])$.
6. *Circuit outputs*: Let $[\![y]\!]$ be the output locally computed by the verifiers. Let $[\![y']\!]$ be the output committed before. Compute the alleged zero $[\![z]\!] = [\![y]\!] - [\![y']\!]$.

V_1 computes $h_1 = H(z_1^1, z_2^1, \cdots, z_s^1)$ and V_2 computes $h_2 = H((-z_1^2), (-z_2^2), \cdots, (-z_s^2))$, where H is a collision-resistant hash function and z_k^1, z_k^2 are the shares of $[\![z_k]\!]$ held by V_1 and V_2 respectively. They send h_1 and h_2 to each other and to the prover, checking if $h_1 = h_2$. If $h_1 \neq h_2$, then P computes h_1 and h_2 from its own shares (it holds all of them) and looks which one was opened incorrectly. Let it be h_k. P broadcasts a complaint against V_k. All shares of V_k are opened using $\mathcal{F}_{transmit}$ and $\mathcal{F}_{pre}$, and the other verifier V_j repeats the computation of V_k.

• **Cheater detection:** At any time, when $\mathcal{F}_{transmit}$ outputs a message $(\mathsf{cheater}, k)$, output $(\mathsf{cheater}, k)$ and stop.

Fig. 7. Real protocol Π_{verify} (verification and cheater detection)

previously committed that output as $[\![y']\!]$ (the output is a message that the prover has sent to another party). To verify the correctness of prover's commitment, the parties produce an alleged zero $[\![z]\!] = [\![y]\!] - [\![y']\!]$.

For particular gate operations, the values $[\![y_i]\!]$ and $[\![z_i]\!]$ are computed as shown in Fig. 6. First, the prover broadcasts to the verifiers some hints, which are just differences between private values and components of the precomputed tuples. Similarly to the pairwise check of Π_{pre}, since each tuple is used only once, all these values come from uniform distribution and can be easily simulated in the security proof. Using these hints and the precomputed tuples, all circuit operations can be reduced to linear combinations of shared values, computed using the homomorphic properties of the sharing scheme.

It is easy to check that, if $z = 0$ for all alleged zeroes z, then $(y_1, \dots, y_l) = op(x_1, \dots, x_k)$ for all gate operations op. The correctness of all broadcast hints is verified using alleged zeroes. The multiplication check is analogous to the pairwise check of Π_{pre}, and for the bit operations, since $y_i \in \{0, 1\}$ (it follows from $b_i \in \{0, 1\}$), the equality $[\![x]\!] = \sum_{i=0}^{m-1} 2^i \cdot [\![y_i]\!]$ implies that $(y_0, \dots, y_{m-1})$ is an m-bit decomposition of x.

So far, all the communication between parties only originates from the prover. Thus the verification of a circuit can be done by the prover first broadcasting a single long message, followed by the verifiers performing local computations.

Checking of alleged zeroes. The verifiers check if $\vec{z} = (z_1, \dots, z_s)$ is equal to $\vec{0}$ similarly to Π_{pre}, exchanging the hashes h_1 and h_2 of shares $\vec{z}^1$ and $(-\vec{z}^2)$.

If $h_1 \neq h_2$, it is still possible that not P, but some V_k has cheated by publishing an incorrect h_k. In this case, h_1 and h_2 are also opened to P, who holds all the shares and hence knows how h_1 and h_2 should look like. P is allowed to complain against one of the verifiers V_k. All the shares of V_k are revealed through $\mathcal{F}_{transmit}$ and $\mathcal{F}_{pre}$. The other verifier V_j can now repeat the computation of V_k and check whether P or V_k was cheating. After this step, V_j knows exactly who the cheater was. Both honest parties now agree on the cheater's identity.

Similarly to the conflict resolving of $\mathcal{F}_{transmit}$, revealing these shares can be easily simulated in UC model since if there is a conflict between P and V_k, then all these shares are already known to the adversary.

All communication in the checking step originates from the verifiers, unless there are complaints. All these messages can be transmitted in the same round. The whole post-execution phase, in the case of no complaints, only requires two rounds of communication. The broadcasts of hints take place in the first round while exchanging the hashes of alleged zero shares takes place during the second round.

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

Lemma 2. *Let the number of parties be $n = 3$. Assuming one corrupted party, there exists a protocol Π_{verify} UC-realizing $\mathcal{F}_{verify}$ in $\mathcal{F}_{pre}$-$\mathcal{F}_{transmit}$-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 Π_{vmpc} of $\mathcal{F}_{vmpc}$, thus proving Theorem. 1 for the 3-party case.

Proposition 1. *Let $n = 3$. Assuming one corrupted party, there exists a protocol Π_{vmpc} UC-realizing $\mathcal{F}_{vmpc}$.*

In Sec. 6, we show how to extend our transformation to the n-party case, proving the general case of Theorem. 1.

5 Evaluation

5.1 Implementation

We have implemented the verification of computations for the Sharemind protocol set [9, 37, 39, 44]. This set covers integer, fix- and floating point operations, as well as shuffling the arrays. Almost all these protocols are generated from a clear description of how messages are computed and exchanged between parties [43], which is very similar to the circuits C^{ℓ}_{ij} defined in Sec. 3.

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 u verified tuples, then (i) they will select μ and κ appropriately for the desired security level (Sec. 4.2); (ii) the prover sends shares of $(\mu u + \kappa)$ 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 κ 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. 4.1). We use SHA-256 as the hash function.

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 $([\![a]\!], [\![b]\!], [\![c]\!])$, both shares of $[\![a]\!]$, both shares of $[\![b]\!]$ and one share of $[\![c]\!]$ are random. The prover only has to send one of the shares of $[\![c]\!]$ 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. The protocols call the networking methods 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 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-256 for that purpose.

5.2 The Cost of Covertly Secure Protocols

For benchmarking, we have chosen the most general protocols of Sharemind over $\mathbb{Z}_{2^{32}}$: multiplication (MULT32), 128-bit AES (AES128), bitwise conjunction (AND32), 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. 236 seconds (Table 1). To verify a single multiplication protocol, we need 6 such triples. We use Sharemind protocol [9, 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 $x_1(y_1+y_2)+x_2y_1$. The amortized preprocessing effort to verify a single 32-bit multiplication is ca. 14µs.

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 236 s. The amortized preprocessing effort to verify a single 128-bit AES block is ca. 2.8ms.

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. 273µs for A2X, and 220µs for X2A.

Table 1. Time to generate $u = 10^8$ verified tuples for $\eta = 80$ ($\mu = 4$, $\kappa = 15000$)

tuple	width	time
Multiplication triples	32 bits	236 s
	64 bits	352 s
Trusted bits	32 bits	72 s
	64 bits	101 s
xor-shared AND triples	32 bits	236 s

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 0.16µs for AND32 and MULT32, 0.04ms for one AES128 block, 2.3µs for A2X, and 5.1µs 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.

Verification. Assuming that all the inputs and the communication have already been committed, and the correlated randomness 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 broad-

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	–	–	–	–	–	–

cast message, and the verifiers exchange the hash of the message to ensure that they got the same message. The results are given in Table 4. When performing 10 million verifications in parallel, the cost of verification is ca. 1.7µs for MULT32 (or AND32), 0.29ms for a single AES128 block, 22µs for A2X32, and 43µs for X2A32.

Table 3. Total amortized cost of covertly secure protocols

	AND	MULT32	AES128	A2X32	X2A32
cost(μs)	0.5	16	3100	297	268

When adding the costs of three phases, we get the results given in Table. 3. These numbers are given for covert security, since we have not applied the verification after *each* round of these protocols. However, as discussed in Sec. 1, since Sharemind protocols are actively private, and no declassifications take place in the intermediate protocol rounds, we achieve active security for given protocols. Hence, it is fair to compare our results with state-of-the-art actively secure protocols.

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. 16µs. This is more than two orders of magnitude faster than any existing solution. For a single AND gate, we get 0.5µs. The total cost of evaluating a 128-bit AES block is ca 3.1ms, 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. 297µs for A2X32 and 268µs for X2A32, and we could not find similar results in related work for comparison.

The recent result of computing AND gates [32] 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). However, our security property is stronger, allowing to pinpoint the cheating party. Our method is also more generic and allows to easily generate precomputed tuples other than multiplication triples, that are useful in verifying protocols other than multiplication.

6 Generalization to n parties

Let the number of parties be n. 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 act as an honest verifier. The largest challenge coming from $n > 3$ is that the corrupted prover P is now able to collaborate with some of the corrupted verifiers V_i. In this section, we show how the building blocks of Sec. 4.1 can be extended to n-party case. We review the definitions of Π_{pre} and Π_{verify}, extending them to n parties.

6.1 Building Blocks

Ensuring message delivery. Assuming an honest majority, the functionality $\mathcal{F}_{transmit}$ of [20] works for any number n of parties. If both the sender and the receiver are corrupted, they are not bound to transmitted messages, and may reveal anything afterwards. However, we may only implement *broadcast with abort* if we use the same approach as we did in the 3-party case.

Table 4. Time and communication of the verification phase

# runs	time (s)								
	AND32	MULT32	AES128	A2X32			X2A32		
				P1	P2	P3	P1	P2	P3
10^1	0.315	0.322	0.472	0.324	0.337	0.323	0.333	0.340	0.337
10^2	0.335	0.337	0.694	0.377	0.387	0.383	0.383	0.413	0.411
10^3	0.387	0.384	1.21	0.496	0.494	0.488	0.465	0.532	0.559
10^4	0.564	0.557	4.46	0.896	0.949	0.930	0.868	1.17	1.21
10^5	0.939	0.952	29.1	2.72	3.08	3.02	2.60	5.31	5.95
10^6	2.72	2.68	–	18.5	21.8	21.4	16.9	37.6	43.0
10^7	16.7	16.7	–	–	–	–	–	–	–

Namely, if the sender is corrupted, then a corrupted receiver may exchange $h_j \neq h_i$ only with *some* parties, while the others think that there are no problems. To solve this, after hashes have been exchanged, the parties will need to wait for a single *broadcast with unanimous abort* round (using e.g. protocols of [40]) within which complaints may be presented. If there is at least one complaint, then broadcast with unanimous abort should be executed on the same message. Opening can be implemented similarly.

Commitments. The commitments can be extended to n parties using any linearly homomorphic (n,t)-threshold sharing scheme. Formally, the prover P is treated as one of the share holders, but in practice P needs to come into play only after all $t-1$ corrupted verifiers have been caught in cheating. For this reason, in the 3-party protocols of Sec. 4, we need to consider only one sharing, the one that does not involve P. Hence, it can be viewed as an instance of $(3,2)$-threshold sharing.

All shares that P sends to V_i are delivered by $\mathcal{F}_{transmit}$. It prevents corrupted parties from tampering with the shares of honest provers, and prevents a corrupted prover from deviating from the shares that it has given to the honest verifiers. If the number of honest parties is at least t, then there is a subset of t verifiers $\mathcal{H}$ that lists only honest parties. In this case, a set of shares can be reconstructed to at most one value. Even if corrupted verifiers collaborate with a corrupted prover and modify their shares later ($\mathcal{F}_{transmit}$ does not commit corrupted parties to each other), this may only lead to inconsistency of shares, and failure to open the commitment. Availability of at least t honest parties allows to maintain the commitment even if all the corrupted parties have left the protocol.

Shamir's sharing is an example of (n,t)-threshold sharing that works over any finite field. For ring operations, replicated secret sharing [16] can be used. We note that the size of shares in the latter case grows exponentially with n.

Preprocessed tuples. Using linear (n,t)-threshold sharing instead of additive, the ideal functionality $\mathcal{F}_{pre}$ can be directly generalized to n parties. Since the sharing scheme is still linear, all the steps of Π_{pre}, up to alleged zero check, can be repeated similarly to the 3-party case, without additional interaction. By properties of (n,t)-threshold sharing, either the shares of z (and also the opened $\hat{a}$ and $\hat{b}$) are inconsistent, or z is equal to the value that has been computed according to the protocol rules from the shares of $\mathcal{H}$. The only difference from the 3-party case is that the verifiers cannot simply exchange the hashes $h_1 = H(z_1^1,\dots,z_s^1)$ and $h_2 = H((-z_1^2),\dots,(-z_s^2))$, and all shares z_j^i need to be broadcast, which is more expensive, especially taking into account that the cost of broadcast itself grows with the number of parties. If the opened shares are inconsistent, the protocol aborts.

6.2 Generalization of Π_{verify}

All the commitments are done using linear (n,t)-threshold sharing instead of additive.

Input commitment. The shares are generated by the prover itself, similarly to the 3-party case. The consistency of shares is not being checked. The commitment is determined by the shares of $\mathcal{H}$ anyway, and it may be only more difficult for the prover to make its proof hold for inconsistent shares.

Randomness commitment. Each verifier V_j first generates $r_j \overset{\$}{\leftarrow} \mathbb{Z}_m$ and commits itself to it by sharing. After V_j has received the shares r_i^j of all the other verifiers V_i, it uses $\mathcal{F}_{transmit}$ to deliver r_i^j to P. After receiving all r_i^j, P reconstructs r_i, and takes $r = \sum_i r_i$. If the shares of some r_i are inconsistent, then all the shares are revealed through $\mathcal{F}_{transmit}$. The cheater is discarded, and the randomness commitment is restarted, this time without the cheater.

It is very important that V_j opens r_i^j to P only *after* it receives r_i^j for all i. This prevents a corrupted P from getting r_j of honest verifiers *before* all corrupted V_i have been committed to r_i that they generated.

Message commitment. During the execution, all messages are transmitted using $\mathcal{F}_{transmit}$, as in the 3-party case. After the execution phase ends, the sender P_s secret-shares the message m_s it had sent to some receiver P_r during the execution, and sends each share m_s^i to V_i using $\mathcal{F}_{transmit}$. At the same time, the receiver P_r shares the message m_r that it actually received, using the same pre-agreed sharing as P_s uses (i.e. such that if $m_s = m_r$, then $m_s^i = m_r^i$ for all i). After receiving m_s^i and m_r^i, V_i checks that $m_s^i = m_r^i$. If they are different, then m_s^i and m_r^i are both revealed through $\mathcal{F}_{transmit}$. If they are indeed different, then either P_s or P_r is corrupted, and the adversary already knows m that was actually transmitted in the execution phase. Hence $\mathcal{F}_{transmit}$ can be used to reveal m.

At this point, both P_s and P_r are committed to the shares of $[\![m]\!]$ that have been issued to the honest parties. It may happen that the sharing $[\![m]\!]$ does not correspond to the m transmitted in the execution phase only if P_s and P_r both are corrupted. In this case, the value of m that was actually transmitted is meaningless anyway, as it can be viewed as an inner value of the joint circuit of P_s and P_r. It is only important that P_s and P_r are committed to the same value. We recall that it is allowed by $\mathcal{F}_{verify}$.

Verification. The first round only involves some broadcasts by the prover, similarly to the 3-party case. On the second round, all the local computations can be done by the verifiers as in the 3-party case, since the linear (n, t)-threshold sharing has the necessary homomorphic properties.

Similarly to Π_{pre}, the verifiers cannot use hashing to verify if $z = 0$, and they need to broadcast all shares of z instead. As in the 3-party case, the shares z^k do not leak any private information of an honest prover. In particular, any share z^k for $k \notin \mathcal{C}$ is uniquely determined by the $t-1$ shares of corrupted parties and the value z, which equals 0.

If $z = 0$, then it should be 0 also if we only take into account the shares z^k of $k \in \mathcal{H}$ that have honestly computed all the linear combinations w.r.t. the commitments. If $z \neq 0$, then it is not clear whether P or some verifier V_i has cheated (or both). In this case, P is allowed to complain about up to $t-1$ verifiers. All the shares of these verifiers are revealed through $\mathcal{F}_{transmit}$, and all the other verifiers repeat their proof steps to recompute their shares z^i. Similarly to the 3-party case, if there is a conflict, then all these shares are known to the adversary anyway, so they can be revealed.

Cheater detection. There are many steps in which a cheater can be detected due to use of $\mathcal{F}_{transmit}$. If one of the corrupted verifiers gets detected during the proof, then we still want the proof to finish, since it is not immediately clear whether P itself is a cheater. In all such cases, the corrupted verifier is discarded from the proof. Using (n, t)-threshold sharing allows the remaining parties to proceed with the proof, even after all $t-1$ corrupted parties have left the protocol.

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 most suitable to be implemented on top of passively secure SMC frameworks that use 3 parties and computation over rings of size 2^m. 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 our method 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 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 [1] 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.

Acknowledgements. The authors are grateful to Emmanuela Orsini for verifying and commenting on the correctness and presentation of proofs. They are also grateful to Aaron Johnson and the anonymous reviewers of PoPETs for their comments. This research has been supported by Estonian Research Council through grant No. IUT27-1, and by the European Regional Development Fund through the Software Technology and Applications Competence Centre (STACC).

References

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

[2] Baum, C., Damgård, I., and Orlandi, C. Publicly auditable secure multi-party computation. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014. Proceedings* (2014), M. Abdalla and R. D. Prisco, Eds., vol. 8642 of *LNCS*, Springer, pp. 175–196.

[3] 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. Proceedings* (2016), M. Manulis, A.-R. Sadeghi, and S. Schneider, Eds., Springer International Publishing, pp. 327–345.

[4] Baum, C., Orsini, E., and Scholl, P. Efficient secure multiparty computation with identifiable abort. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, 2016, Proceedings, Part I* (2016), M. Hirt and A. D. Smith, Eds., vol. 9985 of *LNCS*, pp. 461–490.

[5] Beaver, D. Efficient multiparty protocols using circuit randomization. In *CRYPTO* (1991), J. Feigenbaum, Ed., vol. 576 of *LNCS*, 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, FC 2015, Revised Selected Papers* (2015), R. Böhme and T. Okamoto, Eds., vol. 8975 of *LNCS*, Springer, pp. 227–234.

[7] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., and Talviste, R. Students and taxes: a privacy-preserving study using secure computation. *PoPETs 2016*, 3 (2016), 117–135.

[8] 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 *LNCS*, Springer, pp. 192–206.

[9] 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.

[10] 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 *LNCS*, Springer, pp. 57–64.

[11] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012* (2012), S. Goldwasser, Ed., ACM, pp. 309–325.

[12] Brickell, J., and Shmatikov, V. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT* (2005), B. K. Roy, Ed., vol. 3788 of *LNCS*, Springer, pp. 236–252.

[13] Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium* (2010), pp. 223–239.

[14] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS* (2001), IEEE Computer Society, pp. 136–145.

[15] 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 *LNCS*, Springer, pp. 134–150.

[16] Cramer, R., Damgård, I., and Ishai, Y. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Proceedings* (2005), J. Kilian, Ed., vol. 3378 of *LNCS*, Springer, pp. 342–362.

[17] Cunningham, R., Fuller, B., and Yakoubov, S. Catching MPC cheaters: Identification and openability. Cryptology ePrint Archive, Report 2016/611, 2016. http://eprint.iacr.org/2016/611.

[18] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J. B., and Toft, T. Unconditionally secure constant-rounds multiparty computation for equality, comparison, bits and exponentiation. In *TCC* (2006), S. Halevi and T. Rabin, Eds., vol. 3876 of *LNCS*, Springer, pp. 285–304.

[19] 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 *LNCS*, Springer, pp. 160–179.

[20] 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 *LNCS*, Springer, pp. 128–145.

[21] 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. Proceedings* (2012), I. Visconti and R. D. Prisco, Eds., vol. 7485 of *LNCS*, Springer, pp. 241–263.

[22] 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 *LNCS*, Springer, pp. 1–18.

[23] 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.

[24] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [52], pp. 643–662.

[25] 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/.

[26] Damgård, I., and Zakarias, S. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC* (2013), pp. 621–641.

[27] 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.

[28] 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. [51], pp. 1504–1517.

[29] 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* (2015), The Internet Society.

[30] 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 *LNCS*, Springer, pp. 265–278.

[31] 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, 2015, Proceedings, Part I* (2015), T. Iwata and J. H. Cheon, Eds., vol. 9452 of *LNCS*, Springer, pp. 711–735.

[32] Furukawa, J., Lindell, Y., Nof, A., and Weinstein, O. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology - EUROCRYPT 2017, Proceedings, Part II* (2017), J. Coron and J. B. Nielsen, Eds., vol. 10211 of *LNCS*, pp. 225–255.

[33] 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.

[34] Kamm, L., and Willemson, J. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security* (2014), 1–18.

[35] Katz, J., Ranellucci, S., and Wang, X. Authenticated garbling and efficient maliciously secure multi-party computation. Cryptology ePrint Archive, Report 2017/189, 2017. http://eprint.iacr.org/2017/189.

[36] Keller, M., Orsini, E., and Scholl, P. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM CCS, 2016* (2016), E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, pp. 830–842.

[37] 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.

[38] Kreuter, B., Shelat, A., and Shen, C. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21th USENIX Security Symposium, 2012* (2012), T. Kohno, Ed., USENIX Association, pp. 285–300.

[39] 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. Proceedings* (2014), S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. Yiu, Eds., vol. 8783 of *LNCS*, Springer, pp. 179–197.

[40] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[41] Laud, P., and Pankova, A. Verifiable Computation in Multiparty Protocols with Honest Majority. In *Provable Security - 8th International Conference, ProvSec 2014. Proceedings* (2014), S. S. M. Chow, J. K. Liu, L. C. K. Hui, and S. Yiu, Eds., vol. 8782 of *LNCS*, Springer, pp. 146–161.

[42] Laud, P., and Pettai, M. Secure multiparty sorting protocols with covert privacy. In *Proceedings of Nordsec 2016* (2016).

[43] Laud, P., and Randmets, J. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22nd ACM SIGSAC CCS, 2015* (2015), ACM, pp. 1492–1503.

[44] 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.

[45] Lindell, Y., and Riva, B. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In Ray et al. [51], pp. 579–590.

[46] Mohassel, P., Orobets, O., and Riva, B. Efficient Server-Aided 2PC for Mobile Phones. *Proceedings of Privacy Enhancing Technologies 2016*, 2 (2016), 82–99.

[47] 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 [52], pp. 681–700.

[48] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999), pp. 223–238.

[49] 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* (2015), C. Fournet, M. W. Hicks, and L. Viganò, Eds., IEEE, pp. 75–89.

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

[51] Ray, I., Li, N., and Kruegel, C., Eds. *Proceedings of the 22nd ACM CCS, Denver, CO, USA, October 12-6, 2015* (2015), ACM.

[52] 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 *LNCS*, Springer.

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

[54] Spini, G., and Fehr, S. Cheater detection in SPDZ multiparty computation. In *Information Theoretic Security - 9th International Conference, ICITS 2016, Revised Selected Papers* (2016), A. C. A. Nascimento and P. Barreto, Eds., vol. 10015 of *LNCS*, pp. 151–176.

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

[56] Wang, X., Ranellucci, S., and Katz, J. Authenticated garbling and efficient maliciously secure two-party computation. Cryptology ePrint Archive, Report 2017/030, 2017. http://eprint.iacr.org/2017/030.

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. We now describe their verification. Note that the multiplication protocol only needs multiplications to be verified [9, 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 $\mathbb{Z}_{2^{n+1}}$. Decompose $[\![w]\!]$ into bits and let $[\![y]\!]$ be the highest bit.

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

To compute $[\![y]\!] = [\![x]\!] \gg [\![x']\!]$, the parties first reverse $[\![x]\!]$, using bit decomposition. They shift the reversed value left by $[\![x']\!]$, 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 $([\![r]\!], [\![\vec{s}]\!])$ and $([\![r']\!], [\![\vec{s}']\!])$, the verifiers compute $[\![\hat{r}]\!] = [\![r']\!] - [\![r]\!]$, declassify it, compute $[\![\hat{\vec{s}}]\!] = [\![\vec{s}]\!] - \mathsf{rot}(\hat{r}, [\![\vec{s}']\!])$, declassify it and check that it is a vector of zeroes. Recall (Sec. 4.2) 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 $[\![\vec{y}]\!] = \mathsf{rot}([\![x']\!], [\![\vec{x}]\!])$ for $[\![\vec{x}]\!], [\![\vec{y}]\!] \in \mathbb{Z}_{2^n}^m$ and $[\![x']\!] \in \mathbb{Z}_m$ could be built from bit shifts, but a direct computation is more efficient. The parties need a *rotation tuple* $([\![r]\!], [\![\vec{s}]\!], [\![\vec{a}]\!], [\![\vec{b}]\!])$, where $[\![r]\!]$ and $[\![\vec{s}]\!]$ are a CV tuple (with $r \in \mathbb{Z}_m$ and $\vec{s} \in \mathbb{Z}_{2^n}^m$), $\vec{a} \in \mathbb{Z}_{2^n}^m$ is random and the elements of $\vec{b}$ satisfy $b_i = a_{(i+r) \bmod m}$. The prover broadcasts $\hat{r} = x' - r$ and $\hat{\vec{x}} = \vec{x} - \vec{a}$. The verifiers can now compute

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

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

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

Shuffle. The parties want to apply a permutation σ to a vector $[\![\vec{x}]\!] \in \mathbb{Z}_n^m$, obtaining $[\![\vec{y}]\!]$ satisfying $y_i = x_{\sigma(i)}$. Here $\sigma \in S_m$ is known to the prover and to exactly one of the verifiers [44]. 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 V_1).

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

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

B 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. 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

Table 5. 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*

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 5 presents our findings. For each protocol, the results are presented in the form $\frac{x:y:z}{1:\boldsymbol{a}:b}$. 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 directly 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 the overhead of the online phase of our verification, 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 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 5 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 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 having the overhead of 21 and the protocols working on smaller data having generally smaller overheads.

C Full Security Proofs

We formalize the n-party protocols that have been briefly described in Sec. 6. We use them to construct the protocol Π_{vmpc} UC-realizing $\mathcal{F}_{vmpc}$ that we defined in Sec. 3. We then prove Theorem 2, which is an extension of Theorem. 1 of Sec. 3, that additionally includes the cost estimation of Π_{vmpc}.

Theorem 2. *Let n be the number of parties. Let $\mathcal{C}$ be the set of actively corrupted parties, $|\mathcal{C}| < n/2$. There exists a protocol Π_{vmpc} UC-realizing $\mathcal{F}_{vmpc}$ with correctness error $\varepsilon \le 2^{-\eta}$ for a security parameter η. To achieve this correctness level for verification of r-round protocols, a signature scheme with probability of existential forgery $\delta \le 2^{-\eta-1-\log(7n^2(n+r+3))}$ is assumed.*

In the optimistic mode (as far as no party attempts to cheat), the cost of Π_{vmpc} is the following. Let the local circuits of parties have in total M_x, M_r, M_c, bits of inputs, randomness, and communication respectively, N_b gates requiring bit decompositions, and N_m multiplication gates (we denote $N_g := N_b + N_m$). Let 2^m be the cardinality of the largest used ring. Let the used (n,t)-threshold sharing scheme be such that the bit width of a single share is sh_n times larger than the value itself. The bit communication of different phases has the following upper bounds:

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

The resulting protocol has $9 + r$ rounds, of which 5 come from the preprocessing, 1 from the input commitment, 3 from the verification. In addition, two rounds of broadcast with unanimous abort are executed after the preprocessing and the execution phases, to check whether any party has aborted.

If some corrupted party starts deviating from the protocol, asymptotically, the number of rounds of the execution phase may at most double, and the communication may increase at most $2n$ times. The overhead of the verification phases may be larger, due to use of a more expensive broadcast.

Adversary. Throughout this section, we use $\mathcal{A}$ to denote the adversary that attacks the real protocol Π, and $\mathcal{A}_S$ the adversary that attacks the ideal functionality $\mathcal{F}$. In security proofs, we construct a simulator $\mathcal{S}$ that mediates the communication between $\mathcal{A}$ and $\mathcal{F}$, so that the environment $\mathcal{Z}$ cannot distinguish whether it (together with $\mathcal{A}$) interacts with the protocol Π, or with $\mathcal{F}$ and $\mathcal{S}$. The ideal adversary $\mathcal{A}_S$ can be seen as composition of $\mathcal{A}$ and $\mathcal{S}$. Alternatively, we could put $\mathcal{S}$ in place of $\mathcal{A}_S$, letting it communicate not with $\mathcal{A}$, but directly with $\mathcal{Z}$.

For all ideal functionalities $\mathcal{F}$, we implicitly assume that the inputs of corrupted parties are delivered directly to $\mathcal{A}_S$. For this reason, we will often write that the simulator starts doing something *on input*, meaning the inputs of corrupted parties that are delivered to the simulator by $\mathcal{F}$.

We will often use an informal expression *x is chosen by $\mathcal{A}_S$* in definitions of ideal functionalities, where a message of the form $(command, id, x)$ comes from a corrupted party. Formally, in such cases the ideal functionality $\mathcal{F}$ sends to $\mathcal{A}_S$ a message $(\mathsf{arrived}(command), id, x)$, and waits until $\mathcal{A}_S$ sends back $(\mathsf{change}(command), id, x')$, so that $\mathcal{F}$ will further use x' instead of x. For shortness of presentation, we will avoid writing out this sequence of messages.

In all protocols, the adversary is fully malicious, and the corruptions are static. $\mathcal{C}$ denotes the set of corrupted parties, and $|\mathcal{C}| < n/2$ is assumed in all protocols.

Linear threshold secret sharing. As mentioned in Sec. 6, we use linear (n,t)-threshold sharing for commitments. We write $[\![a]\!] = (\vec{a}^k)_{k\in[n]} = \mathsf{classify}(a)$ to denote the sharing of a, and $a = \mathsf{declassify}((\vec{a}^k)_{k\in[n]})$ to denote the reconstruction of a from shares.

Domains of public mappings. We write $\mathsf{Dom}(f)$ to denote the domain of the mapping f, i.e. the set of arguments on which it is defined. In our protocols, we require e.g. that $\mathsf{Dom}(s) = \mathsf{Dom}(r)$ for the mapping s encoding a sender $s(id)$ and a receiver $r(id)$, thus ensuring that each message identified by id has both the sender and the receiver.

Protocol cost. For measuring protocol cost, we take into account the number of rounds as well as the total number of bits communicated through the network. Formally, we define a type $Cost = \mathbb{N} \times \mathbb{N}$, where the first component is the bit communication, and the second component is the number of rounds. We define the operations $\otimes : Cost \times Cost \to Cost$ (parallel composition) and $\oplus : Cost \times Cost \to Cost$ (sequential composition) as follows:

- $(a,b) \otimes (c,d) = (a + c, \max(b,d))$;
- $(a,b) \oplus (c,d) = (a + c, b + d)$.

We will use the shorthand $(a,b)^{\otimes n}$ to denote $(a,b) \otimes \cdots \otimes (a,b)$, where (a,b) occurs n times. Let the operation $\otimes$ have higher priority than $\oplus$.

C.1 Building Blocks

C.1.1 Transmission, Broadcast, and Opening

We define an ideal functionality $\mathcal{F}_{TR}$ that will be used for communication between the parties. It allows to transmit messages between two parties, forward previously transmitted messages, and also to broadcast and reveal previously transmitted and forwarded messages to all parties. We build Π_{TR} on top of $\mathcal{F}_{transmit}$ of [20], although the efficiency would be improved if we took the internal structure of the protocol $\Pi_{transmit}$ that implements $\mathcal{F}_{transmit}$ and slightly modified it.

The functionality $\mathcal{F}_{TR}$ is given in Fig. 8. Similarly to $\mathcal{F}_{transmit}$ that it given in Fig. 3, each message has a unique identifier id, encoding the sender $s(id)$ and the receiver $r(id)$ of this message, so that all parties know which messages will need to be transmitted between which parties. There is one more party $f(id)$ to which the message should be later forwarded by $r(id)$ (if no forwarding is foreseen, $r(id) = f(id)$). For broadcasts, only $s(id)$ is important, and the values $r(id)$ and $f(id)$ may be undefined. When a message is revealed to all parties, we require that it should be approved by each party of the set $\{s(id), r(id), f(id)\}$, and it should always succeed if at least one of them is honest.

The protocol Π_{TR} implementing $\mathcal{F}_{TR}$ is given in Fig. 9. This protocol is built on top of $\mathcal{F}_{transmit}$ of [20]. As described in Sec. 4.1, to implement broadcast, we let each party use $\mathcal{F}_{transmit}$ to reveal the message to each other party, followed by each pair of parties using $\mathcal{F}_{transmit}$ to reveal to each other the messages they received, checking whether they have received the same message. If the sender is corrupted, this protocol allows that some parties agree on a unique m while the other parties observe inconsistency. Therefore, a more complicated broadcast with unanimous abort (e.g. one of [40]), that we denote $\mathcal{F}_{bc}$, is used to wait for possible complaints. If any party complains, then the message is broadcast again, this time using $\mathcal{F}_{bc}$. The advantage of waiting for complaints instead of using $\mathcal{F}_{bc}$ immediately is that the size of a complaint does not depend on the message size, and in our protocols the messages will be broadcast in large batches. Also, the broadcast of complaints can be postponed to the final round, so that the verification will be repeated again with $\mathcal{F}_{bc}$ if any party does complain.

To implement forwarding, it would be sufficient to let the receiver $P_{r(id)}$ reveal the message m to another party $P_{f(id)}$ using $\mathcal{F}_{transmit}$. However, we want that $P_{f(id)}$ would be able to prove later that the message has been confirmed by $P_{r(id)}$. This requires an additional transmission from $P_{r(id)}$. If $P_{r(id)}$ attempts to transmit not the same message that it has revealed, then it is required to reveal the message to *all* parties, so that all parties will know what message has been delivered to $P_{f(id)}$ by $P_{r(id)}$, and see that it has been approved by $P_{s(id)}$. To ensure that the same message is revealed to all parties, each pair of parties uses $\mathcal{F}_{transmit}$ to reveal to each other the messages they received, similarly to the broadcast.

To reveal a previously forwarded message, the parties first let $P_{s(id)}$ broadcast the message to everyone. If $P_{r(id)}$ or $P_{f(id)}$ disagree, or the broadcast fails, they will use $\mathcal{F}_{transmit}$ to reveal to the other parties the messages that they have actually received. To ensure that the same message is revealed to all parties, each pair of parties uses $\mathcal{F}_{transmit}$ to reveal to each other the messages they received, similarly to the broadcast.

In order to estimate the cost of Π_{TR}, we summarize in Fig. 10 the protocol $\Pi_{transmit}$ of [20] that UC-realizes $\mathcal{F}_{transmit}$. It is based in top of signatures, and we have described it briefly in Sec. 4.1.

Table 6. Costs of different subprotocols of $\Pi_{transmit}$ applied to N-bit messages, using λ-bit signatures

operation	rounds	# bits/op	# bits/rnd.
Cheap mode (if all parties follow the protocol)			
transmit	1	$N + \lambda$	n^2
reveal	1	$N + \lambda$	n^2
Expensive mode (if some party cheats)			
transmit	2	$2(n-1)(N+\lambda)$	0
reveal	2	$2(n-1)(N+\lambda)$	0

Observation 1. *Let λ be the number of bits in a signature. The round and bit communication costs of applying different functions of $\Pi_{transmit}$ to an N-bit message are given in Table 6. The costs of signatures γ_i on $(\mathsf{cheater}, k)$ are counted as one-time overhead that we do not give in the table, since each such overhead may happen only once for P_k and P_i (if P_i has claimed P_k to be corrupted, it will not need to claim that once more). Similarly, although accusation handling can be expensive if any accusations are actually made, we count them as one-time overhead since if a conflict between P_s and P_r occurs once, then all further communication between them takes place in the expensive mode. The n^2 bits per round in the cheap mode are used to let each party inform the other parties whether it wants to complain.*

$\mathcal{F}_{TR}$ 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 $f(id) = r(id)$). For broadcasts, the values of $r(id)$ and $f(id)$ may be undefined.

• **Initialization:** On input (init, s, r, f) from all (honest) parties, where s,r,f are mappings s.t. $\mathsf{Dom}(s) = \mathsf{Dom}(r) = \mathsf{Dom}(f)$, store s, r, f for further use. Deliver (init, s, r, f) to $\mathcal{A}_S$.

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

1. If $s(id) \in \mathcal{C}$, let m be chosen by $\mathcal{A}_S$. $\mathcal{A}_S$ may decide to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).
2. If $s(id) \notin \mathcal{C}$ or $\mathcal{A}$ has not decided to output $(\mathsf{cheater}, s(id))$, output (id, m) to $P_{r(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $r(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.

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

1. If $s(id) \in \mathcal{C}$, let m be chosen by $\mathcal{A}_S$. $\mathcal{A}_S$ may decide to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).
2. If $s(id) \notin \mathcal{C}$ or $\mathcal{A}_S$ has not decided to output $(\mathsf{cheater}, s(id))$, output (id, m) to each party and to $\mathcal{A}_S$.

• **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. For $s(id), r(id) \in \mathcal{C}$, m is chosen by $\mathcal{A}_S$ instead of the value that was actually delivered.
2. If $r(id) \in \mathcal{C}$, $\mathcal{A}_S$ may decide to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).
3. If $r(id) \notin \mathcal{C}$, or $\mathcal{A}$ has not decided to output $(\mathsf{cheater}, s(id))$, output (id, m) to $P_{f(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $f(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.

• **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 $\mathcal{A}_S$. If $s(id), r(id), f(id) \in \mathcal{C}$, then m is chosen by $\mathcal{A}_S$.
$\mathcal{A}_S$ may decide to output $(\mathsf{cheater}, k)$ to all parties for any $k \in \mathcal{C} \cap \{s(id), r(id), f(id)\}$. If $(\mathsf{cheater}, k)$ is output for **all** $k \in \{s(id), r(id), f(id)\}$, then no (id, m) is output to the parties.

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

Assuming that $\mathcal{F}_{transmit}$ has been implemented using $\Pi_{transmit}$, from the definition of Π_{TR}, we can count the number of rounds and the communicated bits of different operations.

Table 7. Costs of different subprotocols of Π_{TR} applied to N-bit messages, using λ-bit signatures

operation	rounds	# bits/op.	# bits/rnd.
Cheap mode (if all parties follow the protocol)			
transmit	1	$N + \lambda$	n^2
broadcast	2	$(n^2 - n)(N + \lambda)$	n^2
forward	1	$2(N + \lambda)$	n^2
reveal	2	$(n^2 - n)(N + \lambda)$	n^2

Observation 2. *Let λ be the number of bits in a signature. The round and bit communication complexities of applying different functions of Π_{TR} to an N-bit message are given in Table 7. We have not specified the precise cost of waiting for complaints, since it does not depend on the message size, and the number of broadcasts in our protocols will depend only on the number of parties.*

Lemma 3. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{TR} UC-realizes $\mathcal{F}_{TR}$ in $\mathcal{F}_{transmit}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{TR}$ described in Fig. 11. It runs local copies of $\mathcal{F}_{transmit}$ and Π_{TR}. We need to show that Π_{TR} can be properly simulated to $\mathcal{A}$, maintaining consistency with inputs and outputs of $\mathcal{F}_{TR}$. Among the other things, since $\mathcal{S}$ does not have control over messages $(\mathsf{cheater}, k)$ that are output by $\mathcal{F}_{TR}$, we need to ensure that $\mathcal{F}_{TR}$ outputs $(\mathsf{cheater}, k)$ to all (honest) parties iff $\mathcal{S}$ simulates the same in Π_{TR}. We explain the simulation of different subprotocols.

Transmission: The emulation of transmission is reduced to $\mathcal{F}_{transmit}$. Any message m that is sent to a corrupted receiver is delivered by $\mathcal{F}_{TR}$ to $\mathcal{S}$, so all messages that are simulated to $\mathcal{A}$ are the same as the corresponding messages output to corrupted receivers by $\mathcal{F}_{transmit}$. Any message m of a corrupted sender is chosen by $\mathcal{A}$, and $\mathcal{S}$ delivers m to $\mathcal{F}_{TR}$, so that the same m is output to the receiver by $\mathcal{F}_{TR}$. The message length $\ell = |m|$ that $\mathcal{S}$ gets from $\mathcal{F}_{TR}$ is used to simulate secure transmissions between honest parties. Hence, the view of $\mathcal{A}$ is consistent with all inputs and outputs of $\mathcal{F}_{TR}$.

In Π_{TR}, 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 (init, s, r, f), where s, r, f are mappings s.t. $\mathsf{Dom}(s) = \mathsf{Dom}(r) = \mathsf{Dom}(f)$, each party stores s, r, f for further use. The parties initialize $\mathcal{F}_{transmit}$ by encoding message senders $\hat{s}(id)$ and receivers $\hat{r}(id)$ as follows.

- For each identifier $id \in \mathsf{Dom}(s)$ for which **forwarding** is foreseen, define an additional identifier id'. Define $\hat{s}(id) = s(id)$, $\hat{r}(id) = \hat{s}(id') = r(id)$, $\hat{r}(id') = f(id)$. If no forwarding is foreseen, then $id = id'$.
- For each identifier $id \in \mathsf{Dom}(s)$ for which either a **forwarding**, a **broadcast** or a **revealing** is foreseen, define $(n-1)$ identifiers id_j for $j \in [n] \setminus \{s(id)\}$, and define $\hat{s}(id_j) = s(id)$, $\hat{r}(id_j) = j$. (For forwarding, these indices will be needed only in the expensive mode).
- Initialize $\mathcal{F}_{transmit}$ with the mappings $\hat{s}$ and $\hat{r}$ encoding the senders and the receivers respectively.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ sends $(\mathsf{transmit}, id, m)$ to $\mathcal{F}_{transmit}$. On input $(\mathsf{transmit}, id)$, each other party sends $(\mathsf{transmit}, id)$ to $\mathcal{F}_{transmit}$. Upon receiving (id, m) from $\mathcal{F}_{transmit}$, $P_{r(id)}$ outputs (id, m) to $\mathcal{Z}$.

• **Broadcast:**

1. On input $(\mathsf{broadcast}, id, m)$ the party $P_{s(id)}$ sends $(\mathsf{transmit}, id_j, m)$ to $\mathcal{F}_{transmit}$ for all $j \in [n] \setminus \{s(id)\}$.
2. On input $(\mathsf{broadcast}, id)$ each party P_k sends $(\mathsf{transmit}, id_j)$ to $\mathcal{F}_{transmit}$ for all $j \in [n] \setminus \{s(id)\}$. Upon receiving (id_k, m) from $\mathcal{F}_{transmit}$, P_k sends $(\mathsf{reveal}, id_k, i)$ to $\mathcal{F}_{transmit}$ for all $i \in [n] \setminus \{s(id), k\}$. On the next round, it expects (id_j, m_j) from $\mathcal{F}_{transmit}$ for all $j \in [n] \setminus \{s(id), k\}$. If $m_k = m_j$ for all j, it outputs (id, m_k) to $\mathcal{Z}$.

If any party receives (id_k, m_k) and (id_j, m_j) s.t. $m_k \neq m_j$, then it broadcasts a complaint using $\mathcal{F}_{bc}$. If a party receives at least one complaint from $\mathcal{F}_{bc}$, it expects that $\mathcal{F}_{bc}$ is executed to broadcast m.

• **Open a received message to all parties:** This step will occur several times, so we describe it separately. Suppose that P_i has transmitted a message identified by id to P_j using $\mathcal{F}_{transmit}$, either during transmission or during forwarding. Now the task of P_j is to reveal m to all other parties, proving that it originated from P_i. As far as there are no complaints, the same 2-round approach as for the broadcast can be used to do the opening. If there have been complaints, we need to use a more complicated protocol:

- P_j sends (reveal, id, k) for all $k \in [n] \setminus \{j\}$ to $\mathcal{F}_{transmit}$.
- Upon receiving (id, m) from $\mathcal{F}_{transmit}$, each party P_k uses $\mathcal{F}_{bc}$ to broadcast m to all parties. P_j itself also uses $\mathcal{F}_{bc}$ to broadcast m to all parties. If m has not been revealed to P_k, it does not broadcast (or broadcasts $\perp$).
- If a party P_k has received at least t broadcast instances of m (even if m has not been revealed to P_k before), it outputs (id, m). Otherwise, it outputs $(\mathsf{cheater}, j)$.

The broadcasts using $\mathcal{F}_{bc}$ ensure that either all parties accept the same m or all of them output $(\mathsf{cheater}, j)$. Requiring at least t confirmations of m ensures that validity of m has been checked by at least one honest party.

• **Forward received message:**

1. On input $(\mathsf{forward}, id)$ if the party $P_{r(id)}$ has already received (id, m) at some point, it sends $(\mathsf{reveal}, id, f(id))$ to $\mathcal{F}_{transmit}$. In addition, $P_{r(id)}$ sends $(\mathsf{transmit}, id', m)$ to $\mathcal{F}_{transmit}$, and each other party sends $(\mathsf{transmit}, id')$ to $\mathcal{F}_{transmit}$. (The same m is delivered to $P_{f(id)}$ twice, but we need it to enable $P_{f(id)}$ to prove later which message $P_{r(id)}$ has forwarded.)
2. On input $(\mathsf{forward}, id)$ the party $P_{f(id)}$ waits for one round and then expects (id, m) and (id', m') from $\mathcal{F}_{transmit}$. If $m = m'$, $P_{f(id)}$ outputs (id, m).

$P_{r(id)}$ may send $m \neq m'$, or refuse to reveal m. In this case, $P_{f(id)}$ broadcasts a complaint, and $P_{r(id)}$ is expected to open the received m to all parties. Now each party knows m, and they may output the same (id, m) on input (reveal, id) later. $P_{f(id)}$ also receives m and outputs (id, m).

• **Reveal received message:** On input (reveal, id), if (id, m) has already been revealed, a party outputs (id, m). Otherwise, the parties act as follows:

1. Let m_s be the message that $P_{s(id)}$ has transmitted before. $P_{s(id)}$ broadcasts m_s to the other parties. All parties have now agreed on the same value m_s.
2. Let m_r be the message that $P_{r(id)}$ has received before. If $P_{r(id)}$ sees that $m_s \neq m_r$, it opens received m_r to all parties.
3. Let $m_f = m'_f$ be the messages that $P_{f(id)}$ has received from $P_{r(id)}$. If $P_{f(id)}$ sees that $m_s \neq m_f$ or $m_r \neq m_f$, it opens received m_f and m'_f to all parties.

All parties have now agreed on the same set of values m_s, m_r, m_f, m'_f. If $m_f = m'_f$, then (id, m_f) is output by each party, regardless of what the other messages are. Otherwise, (id, m_r) is output. The value (id, m_s) is output only in the case when there are problems with m_f, m'_f, m_r or they have not been broadcast. We discuss in the proof why exactly these choices are made.

• **Cheater detection:** At any time when a party receives $(\mathsf{cheater}, k)$ from $\mathcal{F}_{transmit}$, it outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. $\mathcal{F}_{transmit}$ ensures that $(\mathsf{cheater}, k)$ is output to all parties simultaneously.

Fig. 9. Real Protocol Π_{TR}

In $\Pi_{transmit}$, each party works locally with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$.

• **Initialization:** On input (init, s, r), where s, r, are mappings s.t. $\mathsf{Dom}(s) = \mathsf{Dom}(r)$, each party stores s and r for further use. The parties generate their public and secret keys, and exchange their public keys so that the signatures could be verified later.

• **Secure transmit:**

1. *Cheap mode:* use as far as $P_{r(id)}$ does not complain.
   (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 $P_{r(id)}$ 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 $\mathcal{Z}$. Otherwise, it publishes a complaint to ensure that everyone goes to the expensive mode. In [20], cheap exception handling is proposed for publishing complaints, achieving only n^2 additional bits of communication *per round* if there are actually no complaints.
2. *Expensive mode:* a party goes to expensive mode if it receives a complaint from $P_{r(id)}$.
   (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) If a party P_i has received (id, m, σ_s), it sends it to $P_{r(id)}$. Otherwise, it sends a signature γ_i on $(\mathsf{cheater}, s(id))$ 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).

• **Reveal received message:** On input (reveal, id, i), if the party P_j has at some point output (id, m), it sends (id, m, σ_s) to P_i, which outputs (id, m) if σ_s is valid.

• **Cheater detection** In all subprotocols of $\Pi_{transmit}$ we will need a tool for stopping the protocol "gracefully" when corruption is detected. This is done by all parties running the following rules in parallel.

1. At any time when a party P_i sees that a party P_d deviates from the protocol, then P_i signs $(\mathsf{cheater}, d)$ to get signature γ_i and sends the signature to all parties (this happens when P_i does not receive a message from P_j that it was supposed to send).
2. If P_k received a signature γ_i on $(\mathsf{cheater}, d)$ from t distinct parties P_i (the set of γ_i may have been accumulated during several rounds), it considers these as a proof that P_d is corrupted, sends this proof to all parties, outputs $(\mathsf{cheater}, d)$, waits for one round and then terminates all protocols.

Fig. 10. Real Protocol $\Pi_{transmit}$

Broadcast: On all broadcast rounds, for messages moving between the honest parties, $\mathcal{S}$ computes $|m|$ directly from m to simulate transmissions and revealings between them.

For $s(id) \notin \mathcal{C}$, the message m is given to $\mathcal{S}$ by $\mathcal{F}_{TR}$, and we need to ensure that each honest party outputs the same message (id, m). For $s(id) \in \mathcal{C}$, $\mathcal{A}$ chooses messages m_j to deliver to honest P_j. If some of these m_j are different, $\mathcal{S}$ must ensure that all of parties output $(\mathsf{cheater}, s(id))$. Suppose that $s(id)$ has transmitted a message m_j to the party P_j. First, the messages between the parties are exchanged. $\mathcal{S}$ now assumes that parties are waiting for complaints, simulating the broadcast round of $\mathcal{F}_{bc}$. If some party P_k gets $m_j \neq m_k$, or $\mathcal{A}$ decides that a corrupted P_k should complain that $m_j \neq m_k$ has happened, $\mathcal{S}$ simulates execution of $\mathcal{F}_{bc}$. This stronger broadcast either outputs (id, m) to all parties, or, if the sender is cheating, outputs $(\mathsf{cheater}, s(id))$ to all parties. In the latter case, $\mathcal{S}$ delivers $(\mathsf{cheater}, s(id))$ to $\mathcal{F}_{TR}$ to let it output the same. If no complaints come, then each pair of parties has exchanged $m_i = m_j$, and hence they output the same message m. If $s(id) \notin \mathcal{C}$, then it is the same m that was given by $\mathcal{F}_{TR}$. If $s(id) \in \mathcal{C}$, then $\mathcal{S}$ delivers (id, m) to $\mathcal{F}_{TR}$, so that it would output the same message to all parties. Finally, $\mathcal{F}_{TR}$ outputs to all parties the same value that $\mathcal{Z}$ would expect from a real protocol.

Open a received message to all parties: Let m be the message whose revealing $\mathcal{S}$ needs to simulate. An important point of the proof is that, after each party P_j has broadcast its version m_j of the revealed message using $\mathcal{F}_{bc}$, the decision of all parties depends only on these broadcast messages, which are the same for each party, so the final decision of parties is unanimous.

– If $j \notin \mathcal{C}$, then $\mathcal{S}$ simulates revealing m to all parties. It simulates all honest parties, including the sender, broadcasting m using $\mathcal{F}_{bc}$. $\mathcal{A}$ chooses up to $t-1$ messages m_k that P_k for $k \in \mathcal{C}$ should broadcast instead of m. Regardless of messages m_k that $\mathcal{A}$ has chosen for corrupted parties to broadcast, each honest party gets at least t equal messages m broadcast by t honest parties (that is, including the one broadcast by itself). The corrupted parties may open up to $t-1$ messages different from m, so they cannot modify m or cause opening to fail.

• **Initialization:** $\mathcal{S}$ receives (init, s, r, f) from $\mathcal{F}_{TR}$. It initializes a local copy of $\mathcal{F}_{transmit}$.

• **Secure transmit:** $\mathcal{S}$ simulates work of $\mathcal{F}_{transmit}$ on inputs $(\mathsf{transmit}, id)$ and $(\mathsf{transmit}, id, m)$.

- If $s(id) \in \mathcal{C}$ the message m is chosen by $\mathcal{A}$, and $\mathcal{S}$ delivers m to $\mathcal{F}_{TR}$ to report this choice.
- If $s(id) \notin \mathcal{C}$, $r(id) \in \mathcal{C}$, then $\mathcal{S}$ receives the message (id, m) from $\mathcal{F}_{TR}$ and uses it in the simulation.
- If $s(id), r(id) \notin \mathcal{C}$, only $|m|$ is needed for the simulation, and in this case $\mathcal{S}$ gets $(id, |m|)$ from $\mathcal{F}_{TR}$.

• **Broadcast:**

- If $s(id) \notin \mathcal{C}$, $\mathcal{S}$ receives (id, m) from $\mathcal{F}_{TR}$. It takes $m_j = m$ for all $j \in [n] \setminus \{s(id)\}$.
- If $s(id) \in \mathcal{C}$, $\mathcal{S}$ receives m_j for all $j \in [n] \setminus \{s(id)\}$ from $\mathcal{A}$.

$\mathcal{S}$ simulates sending $(\mathsf{transmit}, id_j, m_j)$ and the subsequent $(\mathsf{reveal}, id_j, i)$ to $\mathcal{F}_{transmit}$. If both $s(id), j \in \mathcal{C}$, then $\mathcal{A}$ may reveal arbitrary m_j^*, as allowed by $\mathcal{F}_{transmit}$. $\mathcal{S}$ now simulates waiting for possible complaints using $\mathcal{F}_{bc}$. If each party gets $m_j = m_i$ for all i, j, then all parties have agreed on the same message ($\mathcal{A}$ may still decide that some P_k for $k \in \mathcal{C}$ will complain). If $m_j \neq m_k$ for an honest party P_k, $\mathcal{S}$ simulates its complaint. If there is at least one complaint, $\mathcal{S}$ simulates broadcast of m using $\mathcal{F}_{bc}$.

• **Open a received message to all parties:** Let m be the message whose revealing $\mathcal{S}$ needs to simulate. $\mathcal{S}$ simulates P_j sending (reveal, id, k) for all $k \in [n]$ to $\mathcal{F}_{transmit}$ and the further broadcast of m.

- If $j \notin \mathcal{C}$, then $\mathcal{S}$ simulates revealing m to all parties. It simulates all honest parties broadcasting m using $\mathcal{F}_{bc}$. $\mathcal{A}$ chooses up to $t-1$ messages m_k that P_k for $k \in \mathcal{C}$ broadcast instead of m. In any case, $\mathcal{S}$ simulates honest parties finally outputting m.
- If $j \in \mathcal{C}$, then $\mathcal{A}$ may reveal different messages m_k to different honest parties P_k. In the subsequent broadcast $\mathcal{A}$ chooses up to $t-1$ messages m_k that P_k for $k \in \mathcal{C}$ should broadcast using $\mathcal{F}_{bc}$. $\mathcal{S}$ simulates honest parties outputting m iff the same message m has been broadcast by at least t parties.

• **Forward received message:** In the optimistic setting, the simulation of behaviour of $\mathcal{F}_{transmit}$ on inputs $(\mathsf{transmit}, id', m)$ and $(\mathsf{reveal}, id, f(id))$ is is analogous to secure transmission. $\mathcal{A}$ is allowed to choose m^* to be revealed instead of $m(id)$ only if both $s(id), r(id) \in \mathcal{C}$.
In addition, in the real protocol, $P_{f(id)}$ should check if the transmitted message m and the revealed message m' are the same. $\mathcal{S}$ now simulates waiting for possible complaints using $\mathcal{F}_{bc}$. The messages m and m' can be different only if $r(id) \in \mathcal{C}$, and in this case $\mathcal{S}$ has already received m and m' either from $\mathcal{F}_{TR}$ or from $\mathcal{A}$. If $m \neq m'$, then $\mathcal{S}$ simulates broadcasting m' to each other party, as described in the previous point.

• **Reveal received message:** On input (reveal, id), $\mathcal{S}$ gets m from $\mathcal{F}_{TR}$ and simulates the first broadcast of m_s, waiting for possible complaints using $\mathcal{F}_{bc}$. It simulates the behaviour of honest $P_{r(id)}$ and $P_{f(id)}$, checking if the messages m_r and m_f that they already hold are equal to m_s, simulating revealing m_r or m_f to all parties if necessary. $\mathcal{A}$ may request revealing for $r(id), f(id) \in \mathcal{C}$.
Finally, $\mathcal{S}$ simulates the final choice of honest parties as defined by Π_{TR}.

Fig. 11. Simulator $\mathcal{S}_{TR}$

- If $j \in \mathcal{C}$, then $\mathcal{A}$ may reveal different messages m_k to different honest parties P_k. $\mathcal{S}$ simulates each honest party that received m_k broadcasting it. In order to make a value m accepted, it should have been broadcast by at least t parties. Since there are at most $t-1$ corrupted parties, there is at least one honest party P_j that has also broadcast m, and it would do it only if it received (reveal, id, j) at some point, so there is at least one honest party that has checked that m is indeed a previously transmitted message. If m has been approved by less than t parties, then it may be the case that no honest party accepted m, but in this case all parties unanimously output $(\mathsf{cheater}, j)$.

As the result, $\mathcal{A}$ is convinced that the message m has been revealed to all parties in the real protocol.

Forwarding: Forwarding is reduced to using $\mathcal{F}_{transmit}$ to reveal the message obtained by $P_{p(id)}$ to the party $P_{f(id)}$, and to transmit the same message m to $P_{f(id)}$. For the expensive mode of forwarding, $\mathcal{S}$ needs the message m to simulate resolving the conflict (i.e. simulating revealing m by $P_{r(id)}$ to all other parties). In this case, the value m has already been used by $\mathcal{S}$ before, since the expensive mode is entered only if $r(id) \in \mathcal{C}$ or $f(id) \in \mathcal{C}$. If $r(id) \in \mathcal{C}$, then m has been chosen by $\mathcal{A}$, and $\mathcal{S}$ has delivered (id, m) to $\mathcal{F}_{TR}$, so the same message is stored in the internal state of $\mathcal{F}_{TR}$. If $r(id) \notin \mathcal{C}$, the message (id, m) has come from $\mathcal{F}_{TR}$, and $\mathcal{S}$ has simulated to $\mathcal{A}$ the same m. $\mathcal{S}$ now simulates revealing m to all parties, as described in the previous point. As the result, $\mathcal{F}_{TR}$ outputs to $P_{f(id)}$ the same message m that $\mathcal{A}$ expects the real protocol to output to $P_{f(id)}$.

Revealing messages: $\mathcal{S}$ simulates the behaviour of honest parties as defined by Π_{TR}, simulating the broadcast and the openings whenever necessary. We show that, even if some values of m_s, m_r, m_f, m'_f are different, the parties choose the message that has been provided by the *honest* members of $\{s(id), r(id), f(id)\}$.

- If $m_f = m'_f$, then m_f is output regardless of what the other messages are. Since in $\mathcal{F}_{transmit}$ a message that has been sent by an honest party cannot be modified during revealing, $m_f = m'_f$ proves that both $P_{s(id)}$ and $P_{r(id)}$ have approved this choice.
- If $m_f \neq m'_f$, and m_r has been broadcast, then m_r is output. In this case, it is clear that $P_{f(id)}$ has cheated by opening different messages, and m_r is preferred to m_s since if $P_{s(id)}$ is honest, $P_{r(id)}$ cannot reveal a modified value, and m_r should have been approved by $P_{s(id)}$.
- If $m_f \neq m'_f$ (or at least one of them has not been broadcast), and m_r has not been broadcast, then m_s is output. In this case, $P_{f(id)}$ and $P_{r(id)}$ are either both cheating, or are satisfied by the m_s that $P_{s(id)}$ has broadcast, so m_s is acceptable.

If at least one party in $\{s(id), r(id), f(id)\}$ is honest, then $\mathcal{F}_{TR}$ outputs (id, m) for m that has actually been transmitted or forwarded. $\mathcal{S}$ has ensured that revealing of the same m was simulated to $\mathcal{A}$, since an honest party is always able to justify its version of m. If $\{s(id), r(id), f(id)\}$ are all corrupted, then it is possible for $\mathcal{A}$ to reveal any m^* since it is able to play any behaviour of these parties. In this case, $\mathcal{S}$ delivers (id, m^*) to $\mathcal{F}_{TR}$, so that it outputs (id, m^*) to all parties. In any case, $\mathcal{F}_{TR}$ outputs the message that $\mathcal{Z}$ would expect from a real protocol, based on the view of $\mathcal{A}$.

If forwarding was run in the expensive mode, then no interaction takes place in the real protocol, and hence the behaviour of parties on input (reveal, id) does not need to be simulated. $\mathcal{F}_{TR}$ outputs to all parties the message m that $\mathcal{S}$ used while simulating the expensive mode of forwarding, and $\mathcal{Z}$ expects that the same m will be output by the parties in the real protocol.

Summary: In all cases, the view of corrupted parties that $\mathcal{S}$ simulates to $\mathcal{A}$ is consistent with a real protocol that is executed on the same inputs as $\mathcal{F}_{TR}$, and that produces the same outputs that $\mathcal{F}_{TR}$ does. Hence, no $\mathcal{Z}$ can distinguish the simulated execution from a real protocol. □

Substituting $\mathcal{F}_{transmit}$ with the particular protocol $\Pi_{transmit}$, we get the following corollary.

Corollary 1. *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 Π_{TR} UC-realizes $\mathcal{F}_{TR}$ with correctness error $\varepsilon < N \cdot \delta$ and simulation error 0, where N is the total number of sent messages (including all transmitted, forwarded, broadcast, and revealed messages).*

Proof. 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 sent 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. 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.1.2 Commitments

For commitments, we define an ideal functionality $\mathcal{F}_{share}$. It is given in Fig. 12. For our protocols, it would be sufficient to define a functionality that we could use as a black box for making commitments, computing their linear combinations, and opening them. Since the precomputed tuple generation functionality $\mathcal{F}_{pre}$ (Fig. 4) outputs shares of committed tuples to the parties, and we want to use these shares in another instance of $\mathcal{F}_{share}$ later, we additionally want $\mathcal{F}_{share}$ to output committed values as shares, to open individual shares, and to accept commitments in form of shares.

The ideal functionality. In addition to ordinary commitments (commit) binding a party to a certain value, $\mathcal{F}_{share}$ allows to perform *external commitments* (extcommit) of values that have already been shared among the parties. We will need it to move shares generated by $\mathcal{F}_{pre}$ into a separate instance of $\mathcal{F}_{share}$. A *public commitment* (pcommit) allows $\mathcal{F}_{share}$ to use public values instead of commitments or some of their shares. If the committer $P_{p(id)}$ is corrupted, the work of $\mathcal{F}_{share}$ on input (commit, id) may fail. If it happens, $(\mathsf{cheater}, p(id))$ is output to each party.

$\mathcal{F}_{share}$ allows to compute linear combinations (lc) and truncations $\bmod 2^m$ (trunc) of commitments, and to open them. The *public opening* (open) opens the commitment to all parties. The *private opening* (priv_open) allows to open the value to one party, making that party committed to the value that it receives. Both openings are weak: they may abort, and more expensive methods need to be applied to force opening in such a case.

The *share opening* (share_open) is executed when some particular shares need to be opened. This is a strong opening that results in either opening the value, or causing $(\mathsf{cheater}, k)$ (for a corrupted committer P_k) to be output to all parties.

Each initial commitment, as well as every linear combination computed from the commitments (let us call it a *derived* commitment), is identified by a unique id. The committed values are stored in an array comm as $\mathsf{comm}[id]$. A new session of $\mathcal{F}_{share}$ is initialized when it gets an input (init, m, p) from all (honest) parties, where the quantities m, p are mappings such that $m(id)$ is the bit width of the ring in which the value $\mathsf{comm}[id]$ is committed, and $p(id)$ is the party committed to $\mathsf{comm}[id]$. The formal expression $\mathsf{deriv}[id]$ (represented as a tree whose leaves are identifiers of the initial commitments) shows how $\mathsf{comm}[id]$ has been derived from the initial commitments: we have $\mathsf{deriv}[id] = \mathsf{comm}[id]$ for the initial commitments, and for the derived commitments $\mathsf{deriv}[id]$ describes how each of them has been computed using linear combinations and truncations (an example of a derivation tree is $\mathsf{deriv}[id] = id_1 + 3 \cdot id_2 \pmod{2^{32}}$, where id_1, id_2 are leaves).

In order to simplify the initialization of $\mathcal{F}_{share}$ when it will be called by outer protocols, these mappings have to be initialized only for the initial commitments, and $\mathcal{F}_{share}$ extends them itself to derived commitments. In order to make the extensions uniquely determined, we allow to compute linear combinations only from the values for which $p(id)$ is the same, which means that commitments of different parties cannot be combined.

For simplicity of definition of $\mathcal{F}_{share}$ and the proofs, we assume that $|\mathcal{C}| = t - 1$.

The real protocol. The protocol Π_{share} (Fig. 13-14) implementing $\mathcal{F}_{share}$ works on top of the message transmission functionality $\mathcal{F}_{TR}$ defined in Fig. 9. It uses a linear (n, t)-threshold secret sharing scheme with $t = \lceil n/2 \rceil + 1$. It follows the idea of Sec. 6.1: in order to commit to a value x, the party computes $(x^k)_{k \in [n]} = \mathsf{classify}(x)$ and uses $\mathcal{F}_{TR}$ to deliver x^k to P_k. Since $\mathcal{F}_{TR}$ allows to reveal received messages, such a sharing binds the committer to the t shares that it issued to honest parties, which uniquely define the committed value.

Formally, the committing party P is treated as one of the share holders. However, any subset of t parties that includes P does not commit P to anything, since P is always able to tamper with the share that it holds. In practice, P needs to come into play only after all $t-1$ corrupted share holders have been caught in cheating.

The sharing that was used in the 3-party protocols of Sec. 4 can be seen as an instance of replicated $(3, 2)$-threshold sharing. Applying the definition of replicated secret sharing directly, we would get a share $(0, x_2^P, x_3^P)$ for P, $(x_1^{V_1}, x_2^{V_1}, 0)$ for V_1, and $(x_1^{V_2}, 0, x_3^{V_2})$ to V_2, where $x = x_1^{V_1} + x_1^{V_2} = x_2^P + x_2^{V_1} = x_3^P + x_3^{V_2}$ can be reconstructed by any two parties. After discarding the sharings $x_2^P + x_2^{V_1}$ and $x_3^P + x_3^{V_2}$ that involve the prover, we are left with $x = x_1^{V_1} + x_1^{V_2}$, as it was in Sec. 4.

For a *public commitment*, no interaction between the parties takes place, and the parties locally share x according to a pre-agreed sharing, and store their input shares as $\mathsf{comm}^k[id] \leftarrow x^k$ for further use.

For an *external commitment*, no interaction between the parties takes place, and the parties store their input shares as $\mathsf{comm}^k[id] \leftarrow x^k$ for further use. We note that it does not include the consistency check.

All linear combinations and truncations are computed by parties locally on shares, using linearity of (n, t)-sharing. The truncation is a local operation since 2^m divides $2^{m'}$ whenever $m < m'$.

To open individual shares, the parties use $\mathcal{F}_{TR}$ to reveal $\mathsf{comm}^k[id]$ to all parties. It can be applied only to the initial commitments, since the derived commitments have not been delivered by $\mathcal{F}_{TR}$.

The public opening (open) allows the party $P_{p(id)}$ to open the value x itself by broadcasting all shares x^k. Since x^k is already known to P_k, if it sees that x^k is wrong, it may broadcast a complaint. If any party complains in this way, the opening aborts, and each party gets a set of suspects $\mathcal{K}$ whose guilt has not been proven yet. Since at least t shares of x are held by honest parties, and they uniquely determine the value of x by prop-

$\mathcal{F}_{share}$ works with unique identifiers id, encoding the bit width $m(id)$ of the ring in which the value is committed, and the party $p(id)$ committed to the value. The commitments are stored in an array comm, and their derivations in an array deriv. For each id, the term $\mathsf{deriv}[id]$ is a tree whose leaves are the initial commitments, and the inner nodes are lc, trunc operations applied to them. For the initially committed values, $\mathsf{deriv}[id] = id$. $\mathcal{F}_{share}$ uses a linear (n,t)-threshold sharing scheme with $t = \lceil n/2 \rceil + 1$. The shares of $\mathsf{comm}[id]$ for the initial commitments are stored as $\mathsf{comm}^k[id]$ for $k \in [n]$. Let $\mathcal{H}$ be the set of t honest parties.

• **Initialization:** On input (init, m, p) from all (honest) parties, where $\mathsf{Dom}(m) = \mathsf{Dom}(p)$, store the mappings m, p for further use. Deliver (init, m, p) to $\mathcal{A}_S$.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$ from all (honest) parties, write $\mathsf{comm}[id] \leftarrow x$. Output $(\mathsf{confirmed}, id)$ to all parties and to $\mathcal{A}_S$. Here $\mathsf{comm}[id]$ may already have been defined, and it is rewritten in this case.

On input $(\mathsf{pcommit}, id, j, x^j)$ from all (honest) parties, if $\mathsf{comm}[id]$ has already been defined, write $\mathsf{comm}^j[id] \leftarrow x^j$ and recompute $\mathsf{comm}[id] \leftarrow \mathsf{declassify}(\mathsf{comm}^k[id])_{k\in\mathcal{H}}$. Output $(\mathsf{confirmed}, id)$ to all parties and to $\mathcal{A}_S$. Here $\mathsf{comm}^j[id]$ may already have been defined, and it is rewritten in this case.

• **External Commit:** On input $(\mathsf{extcommit}, id, x^k)$ from each (honest) party P_k, compute $x = \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$ and write $\mathsf{comm}[id] \leftarrow x$, $\mathsf{comm}^k[id] \leftarrow x^k$. Output $(\mathsf{confirmed}, id)$ to all parties and to $\mathcal{A}_S$.

• **Commit:** On input (commit, id, x) from $P_{p(id)}$ and (commit, id) from all (honest) parties:

– If $p(id) \notin \mathcal{C}$, compute $(x^k)_{k\in[n]} = \mathsf{classify}(x)$.
– If $p(id) \in \mathcal{C}$, then $(x^k)_{k\in[n]}$ is chosen by $\mathcal{A}_S$, who may alternatively tell $\mathcal{F}_{share}$ to output a message $(\mathsf{cheater}, p(id))$ to all parties. In this case, take $x = \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$.

Write $\mathsf{comm}[id] \leftarrow x$, $\mathsf{comm}^k[id] \leftarrow x^k$ for all $k \in [n]$, and output (id, x^k) to P_k for $k \in [n]$.
Output the shares of corrupted parties to $\mathcal{A}_S$.

• **Compute Linear Combination:** On input $(\mathsf{lc}, \vec{c}, \vec{id}, id')$ from all (honest) parties, where $|\vec{c}| = |\vec{id}| =: \ell$, $id' \notin \mathsf{Dom}(p)$, and $p' = p(id_i)$ are the same for all $i \in \{1,\dots,\ell\}$, let $m' \leftarrow \min(\{m(id_i) \mid i \in \{1,\dots,\ell\}\})$:

1. Compute $y \leftarrow (\sum_{i=1}^{\ell} c_i \cdot \mathsf{comm}[id_i]) \bmod 2^{m'}$.
2. Write $\mathsf{comm}[id'] \leftarrow y$;
3. Assign $m(id') \leftarrow m'$, $p(id') \leftarrow p'$, $\mathsf{deriv}[id'] \leftarrow \mathsf{lc}(\vec{c}, \vec{id})$.

Syntactic sugar: we write $(id' = \sum_{i=1}^{\ell} c_i \cdot id_i)$ instead of $(\mathsf{lc}, \vec{c}, \vec{id}, id')$.

• **Compute Truncation:** On input $(\mathsf{trunc}, m', id, id')$ from all (honest) parties, where $m(id) \geq m' \in \mathbb{N}$, and $id' \notin \mathsf{Dom}(p)$:

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

Syntactic sugar: we write $(id' = id \bmod 2^{m'})$ instead of $(\mathsf{trunc}, m', id, id')$.

• **Share Open:** On input $(\mathsf{share\_open}, id, k)$ from all (honest) parties, if $\mathsf{comm}[id]$ has been defined, and id corresponds to an initial non-public and non-external commitment, output $(id, k, \mathsf{comm}^k[id])$ to each party and to $\mathcal{A}_S$. Otherwise, output $(id, k, \bot)$ to each party. If both $p(id), k \in \mathcal{C}$, $\mathcal{A}_S$ may choose to output to all parties $(\mathsf{cheater}, p(id))$, or (id, k, m) for any m.

• **Public Open:** On input (open, id) from all (honest) parties, if $(\mathsf{cheater}, k)$ has been output for at least $t-1$ parties, output $(id, \mathsf{comm}[id])$ to all parties and to $\mathcal{A}_S$. Otherwise, output $\mathsf{comm}[id]$ to $\mathcal{A}_S$, who decides whether $(id, \mathsf{comm}[id])$ or $(id, \bot, \mathcal{K}, \mathsf{deriv}[id])$ is output to each party, where $\mathcal{K} \subseteq \mathcal{C}$ if $p(id) \notin \mathcal{C}$, and $\mathcal{K}$ is any set of size at most $t-1$ if $p(id) \in \mathcal{C}$. Alternatively, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, p(id))$ for $p(id) \in \mathcal{C}$.

• **Private Open:** On input $(\mathsf{priv\_open}, id, id')$ from all (honest) parties, if $\mathsf{deriv}[id] \neq id$, output $(id, id', \bot)$. Otherwise, write $\mathsf{comm}[id'] = \mathsf{comm}[id]$, $\mathsf{comm}^k[id'] = \mathsf{comm}^k[id]$ for all $k \in [n]$, output $(id, id', (\mathsf{comm}[id])^k_{k\in[n]})$ to $P_{p(id')}$, and output $(\mathsf{confirmed}, id, id')$ to all parties. If $p(id') \in \mathcal{C}$, output $(id, id', \mathsf{comm}[id])$ to $\mathcal{A}_S$. If $p(id) \in \mathcal{C}$, $\mathcal{A}_S$ may tell $\mathcal{F}_{share}$ to output $(id, id', \bot)$ to each party instead.

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

erties of threshold secret sharing, any attempt of $P_{p(id)}$ to deviate from the commitments results in a complaint from at least one honest party.

If the opening fails, the parties may run *share opening* ($\mathsf{share\_open}$) for each $k \in \mathcal{K}$, using $\mathcal{F}_{TR}$ to reveal the shares of the initial commitments of P_k used in computation of $\mathsf{deriv}[id]$, so that each other party could reconstruct x^k from these shares and see whether $P_{p(id)}$ or P_k was the cheater. Let id' be an identifier of some initial commitment used as a leaf of $\mathsf{deriv}[id]$.

Formally, after the value y^k of $\mathsf{comm}^k[id']$ is revealed, $(\mathsf{pcommit}, id', k, y^k)$ should be input by each party to overwrite shares $\mathsf{comm}^k[id']$ of the initial commitments of parties of $k \in \mathcal{K}$. Finally, after x^k for all $k \in \mathcal{K}$ have been reconstructed in this way, either the commitment x of $P_{p(id)}$ is opened, or $P_{p(id)}$ is publicly blamed by all honest parties (if the shares x^k are inconsistent). Since $\mathcal{F}_{TR}$ does not allow corrupted receivers to reveal incorrect values if the sender was honest, a corrupted P_k cannot make shares of an honest $P_{p(id)}$ inconsistent.

A small caveat here is that is not possible to open shares of external commitments. If the shares have not been sent using $\mathcal{F}_{TR}$, there is no way to check their validity. Solving this problem is delegated to the protocol that embeds $\mathcal{F}_{share}$.

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

Observation 3. *The number of $\mathcal{F}_{TR}$ operations for applying different subprotocols of Π_{share} to an N-bit input is given in Table 8, where tr_M, bc_M, fwd_M, rev_M denote the costs (the number of rounds and the bit communication) of* transmit, broadcast, forward, reveal *respectively on an M-bit message, and sh_n is the number of times the bit width the value shared among n parties is smaller than the bit width of its one share. The notation $\otimes$ and $\oplus$ denotes costs of parallel and sequential execution respectively, as discussed in the beginning of Sec. C.*

We note that, at least with the linear (n, t)-threshold schemes used in this work (Shamir's sharing and replicated sharing), the overhead of share sizes is multiplicative w.r.t. the bit width 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. If $n = 3$, or Shamir's sharing is used, then $\mathsf{sh}_n = 1$.

Table 8. Calls of $\mathcal{F}_{TR}$ for different functionalities of Π_{share} with N-bit values

input	called $\mathcal{F}_{TR}$ functionalities
commit	$\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
open	$\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
share_open	$\mathsf{rev}_{\mathsf{sh}_n \cdot N}$
priv_open	$\mathsf{fwd}^{\otimes n}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
pcommit, extcommit, lc, trunc	–

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 $\mathcal{F}_{TR}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{share}$ described in Fig. 15. The simulator runs a local copy of Π_{share}, together with a local copy of $\mathcal{F}_{TR}$. Let the number of honest parties be $|\mathcal{H}| = t$. Throughout the simulation, the shares $\mathsf{comm}^k[id]$ of $p(id) \in \mathcal{C}$ held by $k \in \mathcal{H}$ should comprise the value $\mathsf{comm}[id]$ held by $\mathcal{F}_{share}$. This ensures that the parties are bound to their commitments.

Transmissions. The delivery of transmitted and broadcast messages is ensured by $\mathcal{F}_{TR}$. At any time when $(\mathsf{cheater}, k)$ message comes from $\mathcal{F}_{TR}$, the party P_k is treated as a cheater by all honest parties unanimously. In this case, $\mathcal{S}$ sends $(\mathsf{cheater}, k)$ to $\mathcal{F}_{share}$, so that it outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$, as it would expect from a real protocol execution.

During public opening, if at least $t - 1$ messages $(\mathsf{cheater}, k)$ have been output to the parties, then $\mathcal{F}_{share}$ does not accept messages $(\mathsf{cheater}, s(id))$ from $\mathcal{S}$ anymore. Since all corrupted parties would have been detected in the real protocol in this case, their opinion would not count, and they could not fail the opening.

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

We show that, in all transmissions, where at least the sender and the receiver is corrupted, $\mathcal{S}$ is able to simulate m in such a way that all outputs of $\mathcal{F}_{share}$ are the same as $\mathcal{Z}$ would expect from a real protocol, observing the view of $\mathcal{A}$.

Commitments. The shares of a corrupted committer are chosen by $\mathcal{A}$. The shares for corrupted receivers $(x^k)_{k \in \mathcal{C}}$ of x are given to $\mathcal{S}$ by $\mathcal{F}_{share}$. $\mathcal{S}$ does not generate any shares itself. This way, for all initial commitments, $\mathcal{A}$ gets the same $t - 1$ shares that are output to corrupted parties by $\mathcal{F}_{share}$.

Openings. On input share_open, $\mathcal{S}$ gets the share from $\mathcal{F}_{share}$ and simulates its broadcast using $\mathcal{F}_{TR}$. On input open, and also priv_open for $p(id') \in \mathcal{C}$, $\mathcal{S}$ needs to simulate to $\mathcal{A}$ all n shares of the value x that is given to $\mathcal{S}$ by $\mathcal{F}_{share}$. For $p(id) \in \mathcal{C}$, $\mathcal{A}$ chooses the shares to broadcast. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulate the shares itself. $\mathcal{S}$ computes the $t - 1$ shares x^k of $k \in \mathcal{C}$ directly from the shares of leaves of $\mathsf{deriv}[id]$ that it has simulated so far. The shares of honest parties are uniquely determined by x and these $t - 1$ shares.

In Π_{share}, each party works locally with unique identifiers id, encoding the bit width $m(id)$ of the ring in which the value is shared, and the party $p(id)$ committed to the value. The parties use a linear (n,t)-threshold sharing scheme with $t = \lceil n/2 \rceil + 1$. Each party stores its own local copy of arrays comm^k for $k \in [n]$, into which it writes the shares known to it. Each party stores a term $\mathsf{deriv}[id]$ (represented by a tree whose leaves are the initial commitments, and the inner nodes are lc, trunc operations applied to them) to remember in which way each $\mathsf{comm}^k[id]$ has been computed. For the initially committed values, let $\mathsf{deriv}[id] = id$.

• **Initialization:** On input (init, m, p) where $\mathsf{Dom}(m) = \mathsf{Dom}(p)$, each party stores m, p for further use. 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 $s(id^{\mathsf{bc}}_k) \leftarrow p(id)$, $r(id^{\mathsf{bc}}_k) \leftarrow \bot$, $f(id^{\mathsf{bc}}_k) \leftarrow \bot$ for the broadcasts (used for share opening). It sends (init, s, r, f) to $\mathcal{F}_{TR}$.

• **Cheater detection:** At any time when a party receives $(\mathsf{cheater}, k)$ from $\mathcal{F}_{TR}$, it outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$, each party computes $(x^k)_{k\in[n]} \leftarrow \mathsf{classify}(x)$ using a pre-agreed sharing, so that different parties would get the same x^k. It writes $\mathsf{comm}^k[id] \leftarrow x^k$ and outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

On input $(\mathsf{pcommit}, id, k, x^k)$, each party writes $\mathsf{comm}^k[id] \leftarrow x^k$ and outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **External Commit:** On input $(\mathsf{extcommit}, id, x^k)$, the party P_k writes $\mathsf{comm}^k[id] \leftarrow x^k$. It outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Commit:** Let id' be the identifier such that the commitment is going to be privately opened to $P_{p(id')}$ later (if no private opening is foreseen, then $id = id'$).

1. On input (commit, id, x), $P_{p(id)}$ shares $(x^k)_{k\in[n]} = \mathsf{classify}(x)$. It sends $(\mathsf{transmit}, id^k_{p(id')}, x^k)$ to $\mathcal{F}_{TR}$, for all $k \in [n]$.
2. On input (commit, id), P_k waits until $(id^k_{p(id')}, x^k)$ comes from $\mathcal{F}_{TR}$.

If all transmissions succeed, each party P_k writes $\mathsf{comm}^k[id] \leftarrow x^k$, $\mathsf{deriv}[id] \leftarrow id$, and outputs (id, x^k) to $\mathcal{Z}$. Otherwise, if $(\mathsf{cheater}, p(id))$ comes from $\mathcal{F}_{TR}$, each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$.

Fig. 13. Real Protocol Π_{share} (init, cheater detection, commitments)

It remains to show that, if the opening succeeds, then the opened values are the same in the real and the simulated execution.

- Let $p(id) \notin \mathcal{C}$. $\mathcal{A}$ may argue against up to $t-1$ shares of $\mathsf{comm}[id]$, causing opening to fail. $\mathcal{S}$ defines the set $\mathcal{K}$ that it sends to $\mathcal{F}_{share}$ in such a way that it contains all parties that have detected inconsistencies with the shares that they hold. The case $\mathcal{K} \not\subseteq \mathcal{C}$ may never happen to an honest party, since no P_k for $k \notin \mathcal{C}$ will present a complaint against an honest $P_{p(id)}$. Hence $\mathcal{K} \subseteq \mathcal{C}$, and it is suitable for $\mathcal{F}_{share}$.
- Let $p(id) \in \mathcal{C}$. We need to show that the equality $\mathsf{declassify}(\mathsf{comm}^k[id])_{k\in\mathcal{H}} = \mathsf{comm}[id]$ is maintained throughout the computation, where $\mathsf{comm}^k[id]$ are the shares held by P_k in the local copy of Π_{share} of $\mathcal{S}$, and $\mathsf{comm}[id]$ 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 $\mathsf{comm}[id]$ are chosen during executing $\mathsf{pcommit}$, $\mathsf{extcommit}$, commit. In all cases, $\mathcal{F}_{share}$ reconstructs directly $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, where x^k for $p(id) \in \mathcal{C}$ are given by $\mathcal{S}$, and for $p(id) \notin \mathcal{C}$ are inputs of $\mathcal{F}_{share}$. In both cases, $\mathcal{S}$ also maintains $\mathsf{comm}^k[id] = x^k$ in the local execution.
  - *Step:* The new values $\mathsf{comm}[id']$ are created by calling lc and trunc. Since both lc and trunc are linear operations, we are using linear secret sharing, and in all truncations we reduce $2^{m'}$ to 2^m for $m < m'$ (so 2^m divides $2^{m'}$), for $f \in \{\mathsf{lc}, \mathsf{trunc}\}$ we have
  $$\begin{aligned} &\mathsf{declassify}(\mathsf{comm}^k[id'])_{k\in\mathcal{H}} \\ &= \mathsf{declassify}(f(\mathsf{comm}^k[id]))_{k\in\mathcal{H}} \\ &= f(\mathsf{declassify}(\mathsf{comm}^k[id]))_{k\in\mathcal{H}} \quad . \end{aligned}$$
  By induction hypothesis, we have $\mathsf{declassify}(\mathsf{comm}^k[id]))_{k\in\mathcal{H}} = \mathsf{comm}[id]$, and since $f(\mathsf{comm}[id]) = \mathsf{comm}[id']$, we have $\mathsf{declassify}(\mathsf{comm}^k[id'])_{k\in\mathcal{H}} = \mathsf{comm}[id']$.

The proof is similar for private opening. The party $P_{p(id')}$ receives x^k directly from P_k. Assuming that no $(id, id', \bot)$ has been output, the shares are consistent, and the value x reconstructed from these shares equals to the value reconstructed from the shares of $\mathcal{H}$. The fact that $P_{p(id')}$ has used $\mathcal{F}_{TR}$ to send x^k back to P_k commits $P_{p(id')}$ to $x' = \mathsf{declassify}(x'^k)_{k\in\mathcal{H}}$. Since each P_k for $k \in \mathcal{H}$ verifies $x^k = x'^k$, both $P_{p(id)}$ and $P_{p(id')}$ are committed to the same value $x' = \mathsf{declassify}(x'^k)_{k\in\mathcal{H}} = \mathsf{declassify}(x^k)_{k\in\mathcal{H}} = x$.

Summary: During the commitments, $\mathcal{S}$ has simulated to $\mathcal{A}$ exactly those shares that are stored in the internal state of $\mathcal{F}_{share}$. Any value that is opened to some party by $\mathcal{F}_{share}$ is the same as the value whose opening is simulated to $\mathcal{A}$. Hence, all inputs and outputs of $\mathcal{F}_{share}$ correspond to what $\mathcal{A}$ would expect from execution of a real protocol, and no $\mathcal{Z}$ can distinguish between the real and the simulated executions. □

• **Compute Linear Combination:** On input $(\mathsf{lc}, \vec{c}, \vec{id}, id')$, where $|\vec{c}| = |\vec{id}| =: \ell$, $id' \notin \mathsf{Dom}(p)$, and $p' = p(id_i)$ are the same for all $i \in \{1, \ldots, \ell\}$, for $m' \leftarrow \min(\{m(id_i) \mid i \in \{1, \ldots, \ell\}\})$, each party P_k

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

• **Compute Truncation:** On input $(\mathsf{trunc}, m', id, id')$, where $m(id) \geq m' \in \mathbb{N}$, and $id' \notin \mathsf{Dom}(p)$, each party P_k

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

• **Share Open:** On input $(\mathsf{share\_open}, id, k)$, each party sends $(\mathsf{reveal}, id^k_{p(id')})$ to $\mathcal{F}_{TR}$. After receiving back $(id^k_{p(id')}, x^k)$, it outputs (id, k, x^k) to $\mathcal{Z}$. The revealing may fail if both $p(id), k \in \mathcal{C}$ or the message has not been transmitted before (if it is a public or an external share), and in this case the parties output $(id, k, \perp)$.

• **Public Open:** On input (open, id):

1. $P_{p(id)}$ takes $x^k \leftarrow \mathsf{comm}^k[id]$ and sends $(\mathsf{broadcast}, id^{\mathsf{bc}}_k, x^k)$ to $\mathcal{F}_{TR}$ for all $k \in [n]$.
2. Upon receiving all shares $(id^{\mathsf{bc}}_k, x^k)$ from $\mathcal{F}_{TR}$, each party P_j compares x^j with the share $\mathsf{comm}^j[id]$ that it holds. If $x^j \neq \mathsf{comm}^j[id]$, it broadcasts (bad, id).

If the shares are inconsistent, or some of them do not arrive, each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$. If at least one message (bad, id) has been broadcast by a party P_k for which no messages $(\mathsf{cheater}, k)$ have come so far, then each party constructs a set $\mathcal{K} := \{k \mid (\mathsf{bad}, id) \text{ was broadcast by } P_k\}$. If $\mathcal{K} \geq t$, then each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$, and otherwise it outputs $(id, \perp, \mathcal{K}, \mathsf{deriv}[id])$ to $\mathcal{Z}$. If there are no problems, then each party reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$ and outputs x to $\mathcal{Z}$.

• **Private Open:** On input $(\mathsf{priv\_open}, id, id')$, if $\mathsf{deriv}[id] = id$:

1. Each party P_k sends $(\mathsf{forward}, id^k_{p(id')})$ to $\mathcal{F}_{TR}$.
2. Upon receiving all $(id^k_{p(id')}, x^k)$ from $\mathcal{F}_{TR}$, $P_{p(id')}$ reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$ and sends $(\mathsf{transmit}, id'^k_{p(id')}, x^k)$ to $\mathcal{F}_{TR}$ to confirm the receipt and make it possible for P_k to prove the confirmation later. If x^k are inconsistent, then $P_{p(id')}$ broadcasts a complaint (bad, id, id').
3. Upon receiving $(id'^k_{p(id')}, x^{*k})$ from $\mathcal{F}_{TR}$, P_k checks if $x^k = x^{*k}$, where x^k is the message that P_k received before the opening. If $x^k \neq x^{*k}$, then P_k broadcasts a complaint (bad, id, id').

If at least one (bad, id, id') is broadcast, or $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{TR}$, all parties output $(id, id', \perp)$. Otherwise, each party P_k writes $\mathsf{comm}^k[id'] \leftarrow \mathsf{comm}^k[id]$, and outputs $(\mathsf{confirmed}, id, id')$ to $\mathcal{Z}$. $P_{p(id')}$ outputs (id, id', x) to $\mathcal{Z}$.

Fig. 14. Real Protocol Π_{share} (local operations and openings)

C.1.3 Coin Tosses

In the protocol Π_{pre} of Sec. 4.2 that generates precomputed tuples, the parties need to agree on common randomness that they will use to decide which tuples will participate in the cut-and-choose step and which tuples will undergo pairwise verification. In this section we describe how the parties agree on that randomness (i.e. toss public coins). The ideal functionality $\mathcal{F}_{coins}$ for generating public randomness is given in Fig. 16.

The protocol implementing $\mathcal{F}_{coins}$, built on top of $\mathcal{F}_{share}$, is given in Fig. 17. The idea behind the randomness generation is quite standard: each party commits to its own random value r_j as an element of $\mathbb{Z}_{2^m}$. After all values have been committed, they are opened, and their sum is used as the randomness (alternatively, the parties could generate random bitstrings of certain length and open their bitwise xor). It is sufficient that only some fixed set of t parties contributes to r_j, and it is only important that at least one of them is honest.

Since in our protocols public randomness is needed to be generated only in the preprocessing phase, before any parties get their inputs, we allow the adversary to abort the execution of $\mathcal{F}_{coins}$ by sending (stop, id) to $\mathcal{F}_{coins}$ at any time.

Lemma 5. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{coins} UC-realizes $\mathcal{F}_{coins}$ in $\mathcal{F}_{share}$-hybrid model.*

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 fixed set of t honest parties. The simulator maintains $\mathsf{declassify}(\mathsf{comm}^k[id])_{k\in\mathcal{H}} = \mathsf{comm}[id]$ for all id s.t $p(id) \in \mathcal{C}$.

• **Initialization**: $\mathcal{S}$ gets (init, m, p) from $\mathcal{F}_{share}$. Based on these values, it initializes its local copy of $\mathcal{F}_{TR}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ should be output for each honest party in Π_{share}, then $\mathcal{S}$ forwards $(\mathsf{cheater}, k)$ to $\mathcal{F}_{share}$, so that $\mathcal{F}_{share}$ would output the same.

• **Public commit:** For $k \in \mathcal{C}$, $\mathcal{S}$ gets x from $\mathcal{F}_{share}$. It computes $(x^k)_{k\in[n]}$ according to the same sharing that the parties of the protocol use, and writes $\mathsf{comm}^k[id] \leftarrow x^k$. If only one share x^k comes from $\mathcal{F}_{share}$, $\mathcal{S}$ writes $\mathsf{comm}^k[id] \leftarrow x^k$ for that share.

• **External commit:** For $k \in \mathcal{C}$, $\mathcal{S}$ gets x^k from $\mathcal{A}$. It writes $\mathsf{comm}^k[id] \leftarrow x^k$.

• **Commit:** For $p(id) \in \mathcal{C}$, $\mathcal{S}$ gets the shares $(x^k)_{k\in[n]}$ from $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ gets the shares $(x^k)_{k\in\mathcal{C}}$ from $\mathcal{F}_{share}$. $\mathcal{S}$ simulates distribution of the shares $(x^k)_{k\in[n]}$ using $\mathcal{F}_{TR}$.
If $(\mathsf{cheater}, k)$ should have come from $\mathcal{F}_{TR}$, $\mathcal{S}$ delivers it to $\mathcal{F}_{share}$. If no $(\mathsf{cheater}, k)$ has come from $\mathcal{F}_{TR}$, then all the shares x^k have been successfully delivered. $\mathcal{S}$ defines $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, in the same way as $\mathcal{F}_{share}$ does.

• **Compute Linear Combination and Truncation:** $\mathcal{S}$ locally performs the computations and assignments for all $k \in \mathcal{C}$. No outputs are produced.

• **Public Open:** $\mathcal{S}$ gets (id, x) from $\mathcal{F}_{share}$. For $p(id) \in \mathcal{C}$, all broadcast shares $(x^k)_{k\in[n]}$ are chosen by $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to generate *all* shares $(x^k)_{k\in[n]}$ by itself. It takes $\mathsf{comm}^k[id']$ for all leaf identifiers id' of $\mathsf{deriv}[id]$, and repeats the computation of $\mathsf{deriv}[id]$ to reconstruct x^k for $k \in \mathcal{C}$. Now it needs to generate the remaining shares x^k for $k \notin \mathcal{C}$ in such a way that $x = \mathsf{declassify}(x^k)_{k\in[n]}$. These shares are uniquely determined by x and the $t-1$ shares of corrupted parties.

$\mathcal{S}$ simulates $(\mathsf{broadcast}, id_k^{\mathsf{bc}}, x^k)$ using $\mathcal{F}_{TR}$. If (bad, id) should be broadcast by any party (either due to bad x^k or by orders of $\mathcal{A}$ for $k \in \mathcal{C}$), then $\mathcal{S}$ orders $\mathcal{F}_{share}$ to output $(id, \bot, \mathcal{K}, \mathsf{deriv}[id])$, where $\mathcal{K}$ is the set of all parties that broadcast (bad, id). If $\mathcal{K} \geq t$, then $\mathcal{S}$ delivers $(\mathsf{cheater}, p(id))$ to $\mathcal{F}_{share}$.

• **Share Open:** If $\mathsf{comm}[id]$ is a public or an external commitment, then all parties output $(id, k, \bot)$ in both the real and the ideal execution. Otherwise, $\mathcal{S}$ simulates opening of x^k:

- If $k \in \mathcal{C}$, then $\mathcal{S}$ has already received x^k either from $\mathcal{F}_{share}$ or $\mathcal{A}$.
- If $k \notin \mathcal{C}$, then $\mathcal{S}$ gets x^k from $\mathcal{F}_{share}$.

$\mathcal{S}$ should simulate sending $(\mathsf{reveal}, id^k_{p(id')})$ to $\mathcal{F}_{TR}$. If both $p(id), k \in \mathcal{C}$, then $\mathcal{A}$ may cause the revealing to fail, and in this case $\mathcal{S}$ orders $\mathcal{F}_{share}$ to output $(id, k, \bot)$. It may also choose to reveal any x^{*k}, and in this case $\mathcal{S}$ orders $\mathcal{F}_{share}$ to output (id, k, x^{*k}).

• **Private Open:**

- If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ simulates opening the shares x^k that have been chosen by $\mathcal{A}$ and have already been delivered to $\mathcal{F}_{share}$.
- If $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulate the shares by itself. There are up to $t-1$ shares issued to P_k for $k \in \mathcal{C}$. $\mathcal{S}$ needs to simulate the remaining shares only if $p(id') \in \mathcal{C}$. In this case, $\mathcal{S}$ gets x from $\mathcal{F}_{share}$, and it computes the remaining x^k in such a way that $x = \mathsf{declassify}(x^k)_{k\in[n]}$, similarly to public open.

For each $k \in \mathcal{C}$, $\mathcal{A}$ chooses the share x^{*k} that should be forwarded by P_k to $P_{p(id')}$. $\mathcal{S}$ simulates all such forwardings using $\mathcal{F}_{TR}$. If $x^{*k} \neq x^k$ for some k, or $p(id') \in \mathcal{C}$ and $\mathcal{A}$ decides to complain, then the broadcast of (bad, id, id') is simulated, and $\mathcal{S}$ delivers $(id, id', \bot)$ to $\mathcal{F}_{share}$.

Fig. 15. The simulator $\mathcal{S}_{share}$

The functionality $\mathcal{F}_{coins}$ works with unique identifiers id, encoding the bit width $m(id)$ of the randomness.

• **Initialization:** On input (init, m), store the mapping m for further use. Deliver m to $\mathcal{A}_S$.

• **Coin toss:** On input (pubrnd, id) from all (honest) parties, generate a random value $r \in \mathbb{Z}_{2^{m(id)}}$. Output (id, r) to each party, and also to $\mathcal{A}_S$.

• **Stopping:** On input (stop, id) from $\mathcal{A}_S$, output $(id, \bot)$ to all parties.

Fig. 16. Ideal functionality $\mathcal{F}_{coins}$

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{coins}$ described in Fig. 18. The simulator runs a local copy of Π_{coins} as well as a local copy of $\mathcal{F}_{share}$.

$\mathcal{S}$ gets (id, r) from $\mathcal{F}_{coins}$. $\mathcal{S}$ should be able to simulate the randomness r_j of honest parties. Up to the last share, $\mathcal{S}$ samples $r_j \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, so these values are distributed uniformly, as $\mathcal{Z}$ expects from a real protocol execution. At most $t-1$ of values r_j are provided by $\mathcal{A}$, so there is at least one r_k left s.t $k \in \mathcal{T} \setminus \mathcal{C}$. This last r_k is computed as $r_k = r - \sum_{j\in\mathcal{T}, j\neq k} r_j$. Since r is distributed uniformly, so is $r - \sum_{j\in\mathcal{T}, j\neq k} r_j$, since even if r_j for $j \in \mathcal{C}$ come from some other distribution, the value r serves as a mask that makes the final share distributed uniformly, as $\mathcal{Z}$ would expect from a real protocol.

The protocol Π_{coins} works with unique identifiers id, encoding the randomness bit width $m(id)$. It uses $\mathcal{F}_{share}$ as a subroutine.

• **Initialization:** On input (init, m), store the mapping m for further use. Let $\mathcal{T}$ be an arbitrary set of fixed t parties. For all $id \in \mathsf{Dom}(m)$, $j \in \mathcal{T}$, for $id_j = (id, j)$, assign $m(id_j) \leftarrow m(id)$, $p(id_j) \leftarrow j$. Send (init, m, p) to $\mathcal{F}_{share}$.

• **Coin toss:** On input (pubrnd, id), each party P_i for $i \in \mathcal{T}$:

1. Generates a random value $r_i \in \mathbb{Z}_{2^{m(id)}}$, and sends $(\mathsf{commit}, id_i, r_i)$ to $\mathcal{F}_{share}$. For $j \neq i$, it sends (commit, id_j) to $\mathcal{F}_{share}$.
2. After receiving $(\mathsf{confirmed}, id_j)$ from $\mathcal{F}_{share}$ for all $j \in \mathcal{T}$, it sends (open, id_j) to $\mathcal{F}_{share}$ for all $j \in \mathcal{T}$.
3. Upon receiving (id_j, r_j) from $\mathcal{F}_{share}$ for all $j \in \mathcal{T}$, it computes $r = \sum_{j \in \mathcal{T}} r_j$ and outputs (id, r) to $\mathcal{Z}$.

• **Stopping:** At any time when $(\mathsf{cheater}, k)$ or $(id, \perp)$ comes from $\mathcal{F}_{share}$, output $(id, \perp)$ to $\mathcal{Z}$.

Fig. 17. The protocol Π_{coins}

• **Initialization:** On input (init, m), $\mathcal{S}$ locally simulates initialization of $\mathcal{F}_{share}$.

• **Coin toss:** On input (pubrnd, id), $\mathcal{S}$ gets (id, r) from $\mathcal{F}_{coins}$. It needs to simulate sending $(\mathsf{commit}, id_j, r_j)$ and (commit, id_j) to $\mathcal{F}_{share}$. The values r_j for $j \in \mathcal{C}$ are provided by $\mathcal{A}$. The values r_j for $j \notin \mathcal{C}$ need to be simulated by $\mathcal{S}$. As far the opening has not started, $\mathcal{A}$ does not expect to see the values r_j generated by $j \notin \mathcal{C}$. Since public opening may start only on all honest parties' agreement, $\mathcal{S}$ is free to wait with their generation until $\mathcal{A}$ commits to r_j for all $j \in \mathcal{C}$. After that, $\mathcal{S}$ generates r_j for $j \notin \mathcal{C}$ in such a way that $\sum_{j \in \mathcal{T}} r_j = r$. More precisely, up to the last share, it samples $r_j \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}$, and then computes the last share as $r_k = r - \sum_{j \in \mathcal{T}, j \neq k} r_j$. Such $k \in \mathcal{T} \setminus \mathcal{C}$ always exists if at least t parties contribute to r_j. For all $j \in \mathcal{T}$, $\mathcal{S}$ simulates opening r_j by sending (open, id_j) to $\mathcal{F}_{share}$.

• **Stopping:** At any time when $(\mathsf{cheater}, k)$ or $(id, \perp)$ comes from $\mathcal{F}_{share}$, $\mathcal{S}$ delivers (stop, id) to $\mathcal{F}_{coins}$.

Fig. 18. The simulator $\mathcal{S}_{coins}$

After $\mathcal{S}$ has decided with all values r_j, it simulates opening them using $\mathcal{F}_{share}$. As described above, $\mathcal{S}$ has generated r_j of $j \notin \mathcal{C}$ in such a way that $\sum_{j \in \mathcal{T}} r_j = r$, so after all r_j are opened and summed up, the final value of r that $\mathcal{A}$ expects to get from the protocol is the same that is output by $\mathcal{F}_{coins}$. Hence, if the protocol does not abort, no $\mathcal{Z}$ can distinguish between the real protocol and the simulated one.

At any time when $(\mathsf{cheater}, k)$ or $(id, \perp)$ should come from $\mathcal{F}_{share}$, $\mathcal{S}$ sends the message (stop, id) to $\mathcal{F}_{coins}$ that causes honest parties to output $(id, \perp)$. Hence, if the protocol fails, it fails simultaneously in the real execution and the simulated one. □

Observation 4. *From the protocol Π_{coins}, we can read out the numbers of $\mathcal{F}_{share}$ operations needed for generating N-bit randomness. The results of Table 8 translate them directly to $\mathcal{F}_{TR}$ operations. Let comm_M, open_M, tr_M, bc_M denote the calls of* commit, open, transmit, broadcast *respectively on an M-bit value. The results are given in Table 9.*

Table 9. Calls of $\mathcal{F}_{share}$ and $\mathcal{F}_{TR}$ (as a subroutine of Π_{share}) using Π_{coins} for generating N-bit randomness

input	$\mathcal{F}_{share}$ calls	respective $\mathcal{F}_{TR}$ calls
pubrnd	$\mathsf{comm}_N^{\otimes t} \oplus \mathsf{open}_N^{\otimes t}$	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n \cdot N}^{\otimes nt}$

C.1.4 Generation of Precomputed Tuples

Fig. 19 depicts the functionality $\mathcal{F}_{pre}$ that we use to generate committed verified precomputed multiplication triples and trusted bits. This is an extension of the functionality of Fig. 4. It supports n parties and allows to generate tuples for different provers and different rings using the same instance of $\mathcal{F}_{pre}$.

In order to share tuple components, and to compute and open their linear combinations and truncations, we will use $\mathcal{F}_{share}$ as a subroutine. The protocol Π_{pre} implementing $\mathcal{F}_{pre}$ is formalized in Fig. 20. A single identifier id corresponds to generating $u(id)$ tuples in a ring $\mathbb{Z}_{m(id)}$ for the prover $P_{p(id)}$. The protocol works similarly to the 3-party case described in Sec. 4. The party $P_{p(id)}$ first generates $\mu \cdot u(id) + \kappa$ tuples itself. Some κ of these tuples are opened using $\mathcal{F}_{share}$ and verified. The other $\mu \cdot u(id)$ tuples are partitioned into $n(id)$ sets, so that one tuple of each set undergoes $(\mu - 1)$ pairwise verifications with the other tuples (all linear combinations and openings that are needed for pairwise verifications are done using $\mathcal{F}_{share}$). Finally, the shares of the remaining $u(id)$ tuples are output by the parties. In order to agree which tuples exactly will be opened and how they will be paired, the parties use $\mathcal{F}_{coins}$ to agree on a public randomness. The arrays $\mathsf{comm}^k[id]$ are used to memorize which shares have been issued to which party, to make it possible to open them later.

$\mathcal{F}_{pre}$ works with unique identifiers id, encoding the bit width $m(id)$ of the ring in which the tuples are shared, the party $p(id)$ that gets all the tuples, and the number $u(id)$ of tuples to be generated. It stores an array comm of already generated tuples.

• **Initialization:** On input (init, m, u, p) from all (honest) parties, where $\mathsf{Dom}(m) = \mathsf{Dom}(u) = \mathsf{Dom}(p)$, store the mappings m, u, p for further use. Deliver m, u, p to $\mathcal{A}_S$. For each id, define a couple of identifiers id_i^k for $k \in [u(id)]$, and $i \in [v]$, where $v = 1$ for trusted bits, and $v = 3$ for triples. Define $\tilde{m}(id_i^k) \leftarrow m(id)$, $\tilde{p}(id_i^k) \leftarrow p(id)$ for all i,k. Send $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{share}$.

• **Trusted bits:** On input (bit, id) from all (honest) parties, if $\mathsf{comm}[id]$ exists, then output $(id, \perp)$ to all parties. Otherwise:

1. Generate a vector of random bits $\vec{b} \overset{\$}{\leftarrow} \mathbb{Z}_2^{u(id)}$. If $p(id) \in \mathcal{C}$, get $\vec{b} \in \mathbb{Z}_2^{u(id)}$ from $\mathcal{A}_S$.
2. Assign $\mathsf{comm}[id] \leftarrow \vec{b}$.
3. Compute $(\vec{b}^k)_{k\in[n]} = \mathsf{classify}(\vec{b})$ over $\mathbb{Z}_{2^{m(id)}}$. Output $(id, \vec{b}^k_{k\in[n]})$ to $P_{p(id)}$, and $(id, \vec{b}^k)$ to P_k for all $k \in [n]$.
   If $p(id) \in \mathcal{C}$, output $(id, \vec{b}^k_{k\in[n]})$ also to $\mathcal{A}_S$. For $k \in \mathcal{C}$, output $(id, \vec{b}^k)$ to $\mathcal{A}_S$. Write $\mathsf{comm}^k[id] = \vec{b}^k$

• **Multiplication triples:** On input (triple, id) from all (honest) parties, if $\mathsf{comm}[id]$ exists, then output $(id, \perp)$ to all parties. Otherwise:

1. Generate $\vec{a} \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}^{u(id)}$, $\vec{b} \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}^{u(id)}$. If $p(id) \in \mathcal{C}$, get $\vec{a}$ and $\vec{b}$ from $\mathcal{A}_S$. Compute elementwise $\vec{c} = \vec{a} \cdot \vec{b}$ in $\mathbb{Z}_{2^{m(id)}}$.
2. Assign $\mathsf{comm}[id] = (\vec{a}, \vec{b}, \vec{c})$.
3. Compute $(\vec{a}^k)_{k\in[n]} = \mathsf{classify}(\vec{a})$, $(\vec{b}^k)_{k\in[n]} = \mathsf{classify}(\vec{b})$, $(\vec{c}^k)_{k\in[n]} = \mathsf{classify}(\vec{c})$ over $\mathbb{Z}_{2^{m(id)}}$.
   Output $(id, (\vec{a}^k_{k\in[n]}, \vec{b}^k_{k\in[n]}, \vec{c}^k_{k\in[n]}))$ to $P_{p(id)}$, and $(id, (\vec{a}^k, \vec{b}^k, \vec{c}^k))$ to P_k for all $k \in [n]$.
   If $p(id) \in \mathcal{C}$, output $(id, (\vec{a}^k_{k\in[n]}, \vec{b}^k_{k\in[n]}, \vec{c}^k_{k\in[n]}))$ also to $\mathcal{A}_S$. For $k \in \mathcal{C}$, output $(id, (\vec{a}^k, \vec{b}^k, \vec{c}^k))$ to $\mathcal{A}_S$. Write $\mathsf{comm}^k[id] = (\vec{a}^k, \vec{b}^k, \vec{c}^k)$.

• **Share Open:** On input $(\mathsf{share\_open}, id, k, j)$ from all (honest) parties, if $\mathsf{comm}^k[id]$ is defined, take the j-th tuple x_j^k of $\mathsf{comm}^k[id]$, and output (id, k, j, x_j^k) to all parties and to $\mathcal{A}_S$. If both $p(id), k \in \mathcal{C}$, then $\mathcal{A}_S$ may choose any message instead of x_j^k.

• **Stopping:** At any time while executing (bit, id) or (triple, id), on input (stop, id) from $\mathcal{A}_S$, stop the functionality and output $(id, \perp)$ to all parties.

Fig. 19. Ideal functionality $\mathcal{F}_{pre}$

Observation 5. *From the description of Π_{pre}, we can extract the total number $\mathsf{nb}_{\mathsf{gen}}(T)$ of bits needed to encode a single tuple of type T, and the numbers $\mathsf{nb}_{\mathsf{op}_1}(T)$ and $\mathsf{nb}_{\mathsf{op}_2}(T)$ of bits to be opened in the pairwise check, where $\mathsf{nb}_{\mathsf{op}_1}(T)$ bits are opened before the last $\mathsf{nb}_{\mathsf{op}_2}(T)$ bits. These values are given in Table 10.*

Table 10. Number of tuple bits involved in different steps (ring cardinality 2^m)

x	$\mathsf{nb}_{\mathsf{gen}}(x,m)$	$\mathsf{nb}_{\mathsf{op}_1}(x,m)$	$\mathsf{nb}_{\mathsf{op}_2}(x,m)$
bit	m	1	m
triple	$3m$	$2m$	m

Lemma 6 (cost of generating tuples using Π_{pre}). *Let $\mathcal{F}_{share}$ be realized by the protocol Π_{share}. Let λ be the number of bits in the randomness seed used by the parties. Given the parameters μ and κ, the number of $\mathcal{F}_{TR}$ operations for generating N tuples of type T using Π_{pre} can be expressed as the quantity prc_T^N defined as*

$$\begin{aligned}\mathsf{prc}_T^N \quad = \quad & \mathsf{tr}^{\otimes n}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)} \otimes (\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda} \oplus \mathsf{bc}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda}) \\ & \oplus (\mathsf{bc}^{\otimes n}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)} \otimes \mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_1}T)} \\ & \otimes \mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_2}T)}) \ .\end{aligned}$$

Proof. From Table. 8, we get the cost $\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n\cdot M}$ of committing an M-bit value using $\mathcal{F}_{share}$, and the cost $\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n\cdot M}$ of opening an M-bit value using $\mathcal{F}_{share}$. The lc operations do not involve any communication. The sum prc_T^N covers all the communication for generating all the N triples of type T.

– $\mathsf{tr}^{\otimes n}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)}$ is the cost of sharing the initial unverified tuples among the n parties in parallel.
– $\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda} \oplus \mathsf{bc}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda}$ is the cost of agreeing on a λ-bit common randomness seed using public opening.
– $\mathsf{bc}^{\otimes n}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)}$ is the cost of cut-and-choose opening. All the κ tuples are opened in parallel.
– $\mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_1}T)}$ and $\mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_2}T)}$ are the costs of the pairwise verifications of all the $(\mu - 1)$ pairs, which counts the total cost of the two openings of this step: the differences between the two tuples, and the alleged zeroes. For trusted bits, the share cost multiplier sh_n can be removed from $\mathsf{nb}_{\mathsf{op}_1}T$ since the difference between two bits is broadcast in plain, not as shares.

Since agreeing on public randomness using Π_{coins} takes more than one round, and the randomness is not opened to any party in the first round, the steps (1) and (2) of Π_{pre} can be done in parallel. Since all communication of

In Π_{pre}, each party works with unique identifiers id, encoding the bit width $m(id)$ of the ring in which the tuples are shared, the party $p(id)$ that gets all the tuples, and the number $u(id)$ of tuples to be generated. Π_{pre} uses $\mathcal{F}_{coins}$ and $\mathcal{F}_{share}$ as subroutines. The parameters μ and κ depend on the security parameter. Let λ be the number of bits in the randomness generator seed.

• **Initialization:** On input (init, m, u, p) from $\mathcal{Z}$, where $\mathsf{Dom}(m) = \mathsf{Dom}(u) = \mathsf{Dom}(p)$, each party stores the mappings m, u, p for further use. For each id, it defines a couple of identifiers id_i^k for $k \in [\mu \cdot u(id) + \kappa]$, and $i \in [v]$, where $v = 1$ for trusted bits, and $v = 3$ for triples. It defines $\tilde{m}(id_i^k) \leftarrow m(id)$, $\tilde{p}(id_i^k) \leftarrow p(id)$ for all i,k. It sends $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{share}$.
In addition, each party defines $\tilde{m}(\mathsf{k}) = 2^{\lambda}$ for some constant identifier k, and sends $(\mathsf{init}, \tilde{m})$ to $\mathcal{F}_{coins}$.

• **Trusted bits:** On input (bit, id):

1. The party $P_{p(id)}$ generates $\mu \cdot u(id) + \kappa$ random bits $b_k \overset{\$}{\leftarrow} \mathbb{Z}_2$. $P_{p(id)}$ sends $(\mathsf{commit}, id_0^k, b_k)$ to $\mathcal{F}_{share}$. Each party sends $(\mathsf{commit}, id_0^k)$ to $\mathcal{F}_{share}$. After getting (id_0^k, B) from $\mathcal{F}_{share}$, each party writes $\mathsf{comm}[id_0^k] = B$, where $B = (b_k^j)_{j\in[n]}$ for $P_{p(id)}$, and $B = b_k^j$ for each other party P_j.
2. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{coins}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices.
3. For $k \in [\kappa]$, each party sends $(\mathsf{open}, id_0^{\pi_k})$ to $\mathcal{F}_{share}$, getting back a bit b_k. If the opening fails, or $b_k \notin \{0, 1\}$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding bits into pairs. Such pairwise verification is repeated $\mu - 1$ times with the same $u(id)$ bits, each time taking the next $u(id)$ indices from π.
   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 party sends $(id_0^{k,k'} = id_0^k - id_0^{k'})$ to $\mathcal{F}_{share}$. If $b_k \neq b_{k'}$ was indicated, each party sends $(id_0^{k,k'} = 1 - id_0^k - id_0^{k'})$ to $\mathcal{F}_{share}$. Each party then sends $(\mathsf{open}, id_0^{k,k'})$ to $\mathcal{F}_{share}$ and checks if the value returned by $\mathcal{F}_{share}$ equals 0. If it does not, then output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ bits in $\mathcal{F}_{share}$. Each party outputs $\mathsf{comm}[id']$ for $id' \in \vec{id}$.

• **Multiplication triples:** On input (triple, id):

1. The party $P_{p(id)}$ generates $\mu \cdot u(id) + \kappa$ random ring element pairs $a_k \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}$, $b_k \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}$. It computes $c_k = a_k \cdot b_k$ for $k \in |\mu \cdot u(id) + \kappa|$. $P_{p(id)}$ sends $(\mathsf{commit}, id_0^k, a_k)$, $(\mathsf{commit}, id_1^k, b_k)$, $(\mathsf{commit}, id_2^k, c_k)$ to $\mathcal{F}_{share}$. Each party sends $(\mathsf{commit}, id_0^k)$, $(\mathsf{commit}, id_1^k)$, $(\mathsf{commit}, id_2^k)$ to $\mathcal{F}_{share}$. After getting (id_0^k, A), (id_1^k, B), (id_2^k, C) from $\mathcal{F}_{share}$, each party writes $\mathsf{comm}[id_0^k] = (A, B, C)$, where $A = (a_k^j)_{j\in[n]}$, $B = (b_k^j)_{j\in[n]}$, $C = (c_k^j)_{j\in[n]}$ for $P_{p(id)}$, and $A = a_k^j$, $B = b_k^j$, $C = c_k^j$ for each other P_j.
2. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{coins}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices.
3. For $k \in [\kappa]$, each party sends $(\mathsf{open}, id_0^{\pi_k})$, $(\mathsf{open}, id_1^{\pi_k})$, $(\mathsf{open}, id_2^{\pi_k})$ to $\mathcal{F}_{share}$, getting back (a_k, b_k, c_k). If the opening fails, or $c_k \neq a_k \cdot b_k$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding triples into pairs. Such pairwise verification is repeated $\mu - 1$ times with the same $u(id)$ triples, each time taking the next $u(id)$ indices from π.
   For each pair (k, k'), let us denote $(id^a, id^b, id^c) = (id_0^k, id_1^k, id_2^k)$, $(id^{a'}, id^{b'}, id^{c'}) = (id_0^{k'}, id_1^{k'}, id_2^{k'})$, $(id^{\hat{a}}, id^{\hat{b}}, id^{\hat{c}}) = (id_0^{k,k'}, id_1^{k,k'}, id_2^{k,k'})$.
   (a) Each party sends $(id^{\hat{a}} = id^a - id^{a'})$, $(id^{\hat{b}} = id^b - id^{b'})$, and then $(\mathsf{open}, id^{\hat{a}})$, $(\mathsf{open}, id^{\hat{b}})$ to $\mathcal{F}_{share}$, getting back $\hat{a}$ and $\hat{b}$ respectively.
   (b) Each party then sends $(id^{\hat{c}} = \hat{a} \cdot id^b + \hat{b} \cdot id^{a'} + id^{c'} - id^c)$ and $(\mathsf{open}, id^{\hat{c}})$ to $\mathcal{F}_{share}$. If $\mathcal{F}_{share}$ returns $\hat{c} \neq 0$, output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ triples in $\mathcal{F}_{share}$. Each party outputs $\mathsf{comm}[id']$ for $id' \in \vec{id}$.

• **Share Open:** On input $(\mathsf{share\_open}, id, k, j)$, each party sends $(\mathsf{share\_open}, id_i^j, k)$ to $\mathcal{F}_{share}$ for all $i \in [v]$, where $v = 1$ for trusted bits and $v = 3$ for triples. After receiving back all (id_i^j, k, x_i^k), the party outputs $(id, k, (x_i^k)_{i\in[v]})$ to $\mathcal{Z}$. The opening of $\mathcal{F}_{share}$ may fail if both $p(id), k \in \mathcal{C}$, and in this case the party outputs $(id, k, j, \perp)$.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ or $(id', \perp)$ for some id' comes from $\mathcal{F}_{share}$ or $\mathcal{F}_{coins}$ while executing (bit, id) or (triple, id), output $(id, \perp)$ to $\mathcal{Z}$.

Fig. 20. Real protocol Π_{pre}

open in Π_{share} originates from the prover, and computing linear combinations using $\mathcal{F}_{share}$ does not introduce any communication, the steps (3) and (4) of Π_{pre} can also be done in parallel. □

Lemma 7. *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 $\mathcal{F}_{coins}$-$\mathcal{F}_{share}$-hybrid model with correctness error $\varepsilon < 2^{\eta}$, and simulation error 0.*

• **Initialization:** On input (init, m, u, p) from $\mathcal{F}_{pre}$, $\mathcal{S}$ initializes internal $\mathcal{F}_{share}$ and $\mathcal{F}_{coins}$.

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

1. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ generates $(\mu - 1)u(id) + \kappa$ additional tuples. It then generates a random seed $s \overset{\$}{\leftarrow} \mathbb{Z}_{2^\lambda}$, and checks which permutation π is generated by s. It then permutes all $\mu \cdot u(id) + \kappa$ tuples (some $u(id)$ of them are "imaginary" tuples that will never be fully simulated by $\mathcal{S}$) in such a way that the $u(id)$ tuples that will be finally left (based on π) are exactly those generated by $\mathcal{F}_{pre}$. It simulates committing them to $\mathcal{F}_{share}$.
   If $p(id) \in \mathcal{C}$, then all the $\mu u(id) + \kappa$ tuples are chosen by $\mathcal{A}$.
2. The parties should jointly agree on a random permutation π of tuples.
   – In order to agree on the same π, $\mathcal{S}$ simulates $\mathcal{F}_{coins}$ in such a way that it provides the same randomness seed s that $\mathcal{S}$ used to rearrange the tuples before the commitments.
   – Now a vector $\vec{s}'$ of certain κ tuples needs to be revealed. $\mathcal{S}$ needs to simulate the public opening of $\mathcal{F}_{share}$, and that requires choosing the values $\vec{s}'$ to be opened. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ takes the $\vec{s}'$ chosen by $\mathcal{A}$ before. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ simulates opening the random valid tuples whose commitment it has simulated before. $\mathcal{S}$ either accepts or rejects the opened 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 $p(id) \in \mathcal{C}$, $\mathcal{S}$ has already received the tuples from $\mathcal{A}$, and it uses them again. If there are any inconsistencies, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{pre}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulate to $\mathcal{A}$ 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 $u(id)$ tuples left for each party that are treated as the final output. For $p(id) \in \mathcal{C}$, $\mathcal{F}_{pre}$ outputs to the parties the shares $\vec{s}$ of 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.

• **Share open:** The share opening is delegated to $\mathcal{F}_{share}$. Unless both $p(id), k \in \mathcal{C}$, it always succeeds since $\mathcal{F}_{share}$ since no message of the form $(\mathsf{extcommit}, id, x^k)$ is being input to $\mathcal{F}_{share}$ by Π_{pre}. $\mathcal{S}$ simulates sending $(\mathsf{share\_open}, id_i^j, k)$ to $\mathcal{F}_{share}$ for all $i \in [v]$, and if $p(id), k \in \mathcal{C}$, then $\mathcal{A}$ may choose to open an arbitrary share that $\mathcal{S}$ delivers to $\mathcal{F}_{pre}$. The shares of $k \notin \mathcal{C}$ are given to $\mathcal{S}$ by $\mathcal{F}_{pre}$ to simulate the opening.

• **Stopping:** If at any time while executing (bit, id) or (triple, id), $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{share}$ or $\mathcal{F}_{coins}$, output (stop) to $\mathcal{F}_{pre}$.

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

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

The simulator will need to generate some non-trivial values during the openings of the cut-and-choose and the pairwise verification, so it should be prepared. During the initial distribution of tuples, for $p(id) \notin \mathcal{C}$, it generates $(\mu - 1)u(id) + \kappa$ additional valid tuples. $\mathcal{S}$ generates a random seed s, computes the permutation π from s, and rearranges the tuples in such a way that exactly those tuples that are chosen by $\mathcal{F}_{pre}$ will be finally left. For $p(id) \in \mathcal{C}$, all $\mu \cdot u(id) + \kappa$ tuples are chosen by $\mathcal{A}$. After $\mathcal{S}$ gets all these tuples from $\mathcal{A}$, it simulates the commitments using $\mathcal{F}_{share}$. For this, it does not need to simulate the values of the tuples of $p(id) \notin \mathcal{C}$ that are output $\mathcal{F}_{pre}$, so there is no inconsistency between the internal state of $\mathcal{F}_{pre}$ and the simulation. The work proceeds as follows.

1. $\mathcal{S}$ initializes $\mathcal{F}_{coins}$ and simulates its run in such a way that the seed s will be the same that was used by $\mathcal{S}$ in the beginning. Since $\mathcal{S}$ has generated s uniformly, this is what $\mathcal{Z}$ expects to get from $\mathcal{F}_{coins}$ in the real protocol.
2. For cut-and-choose of honest provers, $\mathcal{S}$ generates the opened tuples from the same distribution as an honest prover would. By choice of the randomness seed s, these tuples are not related to the $u(id)$ tuples generated by $\mathcal{F}_{pre}$, and hence for $\mathcal{Z}$ they seem as coming from the real protocol.
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, since they are not included into $\mathcal{F}_{pre}$ anyway. The most interesting is the last μ-th iteration. Let k be the tuple that has been chosen by $\mathcal{F}_{pre}$ and has not been shown to $\mathcal{S}$, and let k' be the tuple that $\mathcal{S}$ may still choose itself.
   (a) *Trusted bits:* First of all, $\mathcal{S}$ 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 $1 - b_k - b_{k'}$.

For an honest prover, that value is always 0 since it tells honestly whether $b_k \neq b_{k'}$.

(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 $c_{k'} = a_{k'} \cdot b_{k'}$.

If the protocol succeeds for $p(id) \notin \mathcal{C}$, the finally left $u(id)$ tuples are exactly those that are generated by $\mathcal{F}_{pre}$, so the outputs are the same in the real and the simulated executions, and since $\mathcal{A}$ has received up to $t-1$ uniformly distributed shares of remaining tuples that have not been opened, no $\mathcal{Z}$ can distinguish between these executions. For $p(id) \in \mathcal{C}$, these $u(id)$ tuples are all generated by $\mathcal{A}$. If any of these tuples is invalid, there will be immediate difference with $\mathcal{F}_{pre}$ that outputs to the parties only valid tuple shares. We show that, if finally $u(id)$ 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, $1-b_k-b_{k'}$ is output. If indeed $1-b_k-b_{k'} = 0$, 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 $a_k \cdot b_k = c_k$.

We have shown that the only possibility for the prover to cheat is to put two invalid tuples into the same pair during the pairwise verification. 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 $u := u(id)$. If $P_{p(id)}$ wants to have ℓ bad tuples among the final u ones, it has to do the following:

1. put $\mu \cdot \ell$ bad tuples into the initial set of $(\mu \cdot u + \kappa)$ 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, from the point of view of a malicious prover, the probability of steps (2) and (3) succeeding is less than $2^{-\eta}$ for a security parameter η. We note that analogous results are achieved by [32],and their bounds are even better.

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

$$\begin{aligned} P_{\text{c\&c}}(\mu, u, \kappa, \ell) &= \frac{\binom{\mu(u-\ell)+\kappa}{\kappa}}{\binom{\mu u+\kappa}{\kappa}} \qquad (1)\\ &= \frac{(\mu(u-\ell)+\kappa)!}{(\mu u+\kappa)!} \cdot \frac{(\mu u)!}{(\mu(u-\ell))!} \\ &= \frac{\mu u \cdots (\mu(u-\ell)+1)}{(\mu u+\kappa)\cdots(\mu(u-\ell)+\kappa+1)} \\ &\leq \left(\frac{\mu u}{\mu u+\kappa}\right)^{\mu\cdot\ell} . \end{aligned}$$

In particular, if $\ell \geq cu$ for some $c \in [0,1]$, then, assuming $\kappa \leq \frac{\mu u}{2}$,

$$\begin{aligned} P_{\text{c\&c}}(\mu, u, \kappa, \ell) \leq \left(\frac{\mu u}{\mu u+\kappa}\right)^{\mu u c} &= \left(\frac{1}{1+\frac{\kappa}{\mu u}}\right)^{\mu u c} \\ &= \frac{1}{\left(1+\frac{\kappa}{\mu u}\right)^{\frac{\mu u}{\kappa}\cdot c\kappa}} \\ &\leq \frac{1}{2^{c\kappa}} . \qquad (2) \end{aligned}$$

Probability of passing pairwise checking. For the pairwise checking, we partition the μu tuples into u groups of size μ, so that only one tuple of each group is left after checking. We have $\binom{\mu u}{\mu}$ ways to select the first group, $\binom{\mu u-\mu}{\mu}$ ways to select the second group, $\binom{\mu u-2\mu}{\mu}$ 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 $u!$. The number of groupings of μu tuples into u groups is

$$\begin{aligned}\mathcal{G}(\mu,u) &= \frac{1}{u!}\prod_{i=0}^{u-1}\binom{\mu(u-i)}{\mu} \qquad (3)\\ &= \frac{1}{u!}\left(\frac{1}{\mu!}\right)\prod_{i=0}^{u-1}\frac{(\mu(u-i))!}{(\mu(u-i-1))!}\\ &= \frac{1}{u!}\left(\frac{1}{\mu!}\right)^{u}\frac{(\mu(u-0))!}{(\mu(u-(u-1)-1))!}\\ &= \frac{(\mu u)!}{u!(\mu!)^u}\ .\end{aligned}$$

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 $\mathcal{G}(\mu,\ell)\cdot\mathcal{G}(\mu,u-\ell)$, and thus the probability of passing the pairwise check is

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

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

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

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

$$\begin{aligned}P_{\mathrm{pp}}(\mu,u,\kappa,1) &= \binom{u}{1}/\binom{\mu u+k}{\mu\cdot 1}\le u/\left(\frac{\mu u+k}{\mu}\right)^{\mu}\\ &= u/\left(u+\frac{k}{\mu}\right)^{\mu}\le u^{1-\mu}\ .\end{aligned}$$

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

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

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

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

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

getting a sufficient lower bound L_κ for κ:

$$\begin{aligned}L_\kappa &= u^{1/\mu}(\mu\ell+1)-1-\mu(u-\ell-1)\\ &= \mu(u^{1/\mu}\ell-u+\ell)+u^{1/\mu}+\mu-1\\ &\le \mu(u^{1/\mu}cu-u+cn)+u^{1/\mu}+\mu-1\\ &= \mu u(c(u^{1/\mu}+1)-1)+u^{1/\mu}+\mu-1\ .\end{aligned}$$

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

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

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

$$\begin{aligned}\eta(u^{1/\mu}+\mu-1) &\le u^{1/(1+\eta/\log n)}+\mu-1\\ &\le u^{\log n/\eta}+\mu-1\\ &= 2^{-\eta}+\mu-1\\ &\le \mu\le\frac{\mu u}{2}\end{aligned}$$

for $u\ge 2$, and

$$\begin{aligned}\eta(u^{1/\mu}+1) &\le \eta(u^{1/(1+\eta/\log n)}+1)\\ &\le \eta(u^{\log n/\eta}+1)\\ &= \eta(2^{-\eta}+1)\\ &\le \frac{3}{2}\eta\ .\end{aligned}$$

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

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

$$\mu \geq 1 + \eta / \log u \ ,$$
$$\kappa \geq \max\{(u^{1/\mu} + 1)\eta, u^{1/\mu} + \mu - 1\} \ .$$

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 $\kappa \geq \max\{(N^{1/\mu} + 1)\eta, N^{1/\mu} + \mu - 1\}$. □

C.2 Verification of Circuit Computation

The ideal functionality $\mathcal{F}_{verify}$ for verification of circuit computation has been given in Figure 2. It allows to verify computations w.r.t. committed inputs, outputs, randomness, and communicated messages. Figure 22 depicts essentially the same functionality, but it uses a different notation. In particular, instead of taking circuits directly as inputs, it takes a set of subcircuits with fan-out 1 (we call such a subcircuit a *function f*), thus verifying the circuit outputs one by one. Such representation makes handling of identifiers easier.

The protocol Π_{verify} implementing $\mathcal{F}_{verify}$ is given in Fig. 23-24. It works on top of the functionality $\mathcal{F}_{share}$ depicted in Fig. 12 (used to commit inputs, randomness, and communication), and the precomputed tuple generation functionality $\mathcal{F}_{pre}$ depicted in Fig. 19 (used to generate tuples for verification). In order to combine shares of these functionalities, the parties insert the shares that they get from $\mathcal{F}_{pre}$ to $\mathcal{F}_{share}$ using its interface extcommit. All linear combinations, truncations and openings that are related to the verification are done using $\mathcal{F}_{share}$ only. The functionality $\mathcal{F}_{pre}$ may need to be called again if opening of $\mathcal{F}_{share}$ fails, so that the external commitments could be published using the help of $\mathcal{F}_{pre}$.

We will need to generate shared randomness that is known only to a certain party. The protocol that we use for it is very similar to Π_{coins} of Fig. 17. It also uses $\mathcal{F}_{share}$, and the difference is that the randomness is opened only to one party. Using the protocol Π_{share} for implementing $\mathcal{F}_{share}$, we see that each r_j of the sum $r = r_1 + \cdots + r_n$ is in turn shared to $(r_j^1, \ldots, r_j^n)$ using linear (n,t)-threshold sharing. In this way, we get the same protocol for randomness generation that has been described in Sec. 6.2. Similarly to Π_{coins}, having just t values r_j is sufficient, and it is only important that at least one r_j in the sum is generated by an honest party. If we apply the same approach to $n = 3$, we have $t = 2$, and $r = r_1 + r_2$, where in turn $r = r_1^1 + r_1^2$ and $r_2 = r_2^1 + r_2^2$, where the share r_j^k is meant for the verifier V_j. Since V_k already holds r_k^k and it knows r_k anyway, it may just take $r_k^k = r_k$ and let $r_j^k = 0$, so that r_j^k would not need to be sent to V_j. Each verifier V_k only needs to send r_k to P, and we get the 3-party randomness generation that was given in Sec. 4.3.

Both $\mathcal{F}_{verify}$ and Π_{verify} use unique identifiers id. Each such identifier corresponds to some wire of the circuit that is being verified. It encodes the two parties $p(id)$ and $p'(id)$ (possibly $p(id) = p'(id)$) committed to a particular valuation $\mathsf{comm}[id]$ of the wire, and a function $f(id) =: f'$, describing the computation of a single circuit output, with its input identifiers $\vec{xid}(id)$, so that the parties may verify whether $\mathsf{comm}[id] = f'((\mathsf{comm}[i])_{i \in \vec{xid}(id)})$. If the computation of $\mathsf{comm}[id]$ is not needed to be verified (i.e. it is some input commitment), then formally $f(id) = \mathsf{id}_R$ (identity over some ring R), and $\vec{xid}(id) = []$.

Observation 6. *From the definition of Π_{verify}, we extract the number of different tuples required for each operation type directly from the description of the initialization phase. They are given in Table 11.*

Table 11. Number of precomputed tuples for basic operations

operation	type	# tuples	# bits
Linear combination	–	–	–
Conv. to a smaller ring	–	–	–
Bit decomp. in $\mathbb{Z}_{2^m}$	bit	m	m
Multiplication in $\mathbb{Z}_{2^m}$	triple	1	m
Extend $\mathbb{Z}_{2^{m_x}}$ to $\mathbb{Z}_{2^{m_y}}$	bit	m_x	m_y

Lemma 8 (cost of initializing Π_{verify}). *Let Π_{verify} use the implementation Π_{pre} of $\mathcal{F}_{pre}$ with λ-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), and N_m multiplications. Let 2^m be the cardinality of the largest ring involved in the computation. The cost of initializing Π_{verify} is upper bounded by*

$\mathcal{F}_{verify}$ works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $\mathsf{comm}[id]$, the function $f(id)$ to verify, and the input identifiers $\vec{xid}(id)$ on which $f(id)$ should be verified w.r.t. the output identified by id.

• **Initialization:** On input $(\mathsf{init}, f, \vec{xid}, p, p')$ from all (honest) parties, where $\mathsf{Dom}(f) = \mathsf{Dom}(\vec{xid}) = \mathsf{Dom}(p) = \mathsf{Dom}(p')$, store the mappings f, $\vec{xid}$, p, p' for further use. Deliver $(\mathsf{init}, f, \vec{xid}, p, p')$ to $\mathcal{A}_S$. If $\mathcal{A}_S$ responds with (stop), output $\perp$ to all parties.

• **Input Commitment:** On input $(\mathsf{commit\_input}, id, x)$ from $P_{p(id)}$, and $(\mathsf{commit\_input}, id)$ from all (honest) parties, if $\mathsf{comm}[id]$ it not defined yet, assign $\mathsf{comm}[id] \leftarrow x$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$.

• **Message Commitment:** On input $(\mathsf{send\_msg}, id, x)$ from $P_{p(id)}$ and $(\mathsf{send\_msg}, id)$ from all (honest) parties, output x to $P_{p'(id)}$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$. If $p'(id) \in \mathcal{C}$, output x to $\mathcal{A}_S$. Assign $\mathsf{sent}[id] \leftarrow x$.

On input $(\mathsf{commit\_msg}, id)$ from all (honest) parties, check if $\mathsf{sent}[id]$ and $\mathsf{comm}[id]$ are defined. If $\mathsf{sent}[id]$ is defined, and $\mathsf{comm}[id]$ is not defined, assign $\mathsf{comm}[id] \leftarrow \mathsf{sent}[id]$. If both $p(id), p'(id) \in \mathcal{C}$, assign $\mathsf{comm}[id] \leftarrow x^*$, where x^* 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 $\mathsf{comm}[id]$ is not defined yet, generate a fresh randomness r in $\mathbb{Z}_{2^m}$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. If $\mathcal{A}_S$ sends (stop), output $\perp$ to all parties. Otherwise, assign $\mathsf{comm}[id] \leftarrow r$. For $p(id) \in \mathcal{C}$, deliver r to $\mathcal{A}_S$.

• **Verification:** On input (verify, id) from all (honest) parties, if $\mathsf{comm}[id]$ and $\mathsf{comm}[i]$ have been defined for all $i \in \vec{xid}(id)$, take $\vec{x} \leftarrow (\mathsf{comm}[i])_{i \in \vec{xid}(id)}$ and $y \leftarrow \mathsf{comm}[id]$. For $f \leftarrow f(id)$, compute $y' \leftarrow f(\vec{x})$. If $y' - y = 0$, output $(id, 1)$ to each party. Otherwise, output $(id, 0)$ to each party. Output the difference $y' - y$, to $\mathcal{A}_S$.

• **Cheater detection:** On all inputs involving id, if $p(id) \in \mathcal{C}$, $\mathcal{A}_S$ may input $(\mathsf{cheater}, p(id))$. In this case, $\mathsf{comm}[id]$ and $\mathsf{sent}[id]$ are not assigned, $(\mathsf{cheater}, p(id))$ is output to each party. If $\mathcal{A}_S$ does not input $(\mathsf{cheater}, p(id))$, then each commitment ends up outputting $(\mathsf{confirmed}, id)$ to each party. If $(\mathsf{cheater}, p(id))$ comes from $\mathcal{A}_S$ during the execution of (verify, id), then $\mathcal{F}_{verify}$ outputs $(id, 0)$ to all parties instead of $(\mathsf{cheater}, p(id))$.

Fig. 22. Ideal functionality $\mathcal{F}_{verify}$

$$\begin{aligned}\mathsf{vcost}_{pre}^{N_b,N_m,m} \quad = \quad & \mathsf{tr}^{\otimes n}_{\mathsf{sh}_n\cdot m\cdot((\mu(N_b\cdot m+3N_m)+\kappa(m+3))} \\ & \otimes(\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda} \oplus \mathsf{bc}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda}) \\ & \oplus\ (\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n\cdot m\cdot\kappa(m+3)} \\ & \otimes\ \mathsf{bc}^{\otimes n}_{(\mu-1)m(N_b+\mathsf{sh}_n\cdot 2N_m)} \\ & \otimes\ \mathsf{bc}^{\otimes n}_{\mathsf{sh}_n\cdot(\mu-1)m(N_bm+N_m)}) \ .\end{aligned}$$

Proof. The number of different tuples used by each operation is given in Table. 11. By Lemma 6, the cost of generating N tuples of type x over a ring of size 2^m is

$$\begin{aligned}\mathsf{prc}_T^N \quad = \quad & \mathsf{tr}^{\otimes n}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)} \otimes (\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda} \oplus \mathsf{bc}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda}) \\ & \oplus\ (\mathsf{bc}^{\otimes n}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{gen}}T)} \otimes\ \mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_1}T)} \\ & \otimes\mathsf{bc}^{\otimes n}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_2}T)}) \ .\end{aligned}$$

where $T = (x, m)$, and the definitions of subterms can be found in Table 10.

The total number of the transmitted and broadcast bits is linear in the terms $N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{gen}}(x, m)$, $N \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m)$, and $N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m)$. Hence it suffices to find the upper bounds for these three quantities. 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 3).

– The bit decomposition and the conversion to a larger ring both require m trusted bits of m bits each. As shown in the proof of Lemma 6, the multiplier sh_n can be removed from $\mathsf{nb}_{\mathsf{op}_1}(\mathsf{bit}, m)$ since the prover broadcasts the difference bit in plain. Hence for the bit-related gates we have

$$\begin{aligned}N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{gen}}(x, m) \quad & \leq \quad N_b \cdot m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{gen}}(\mathsf{bit}, m) \\ & = \quad N_b \cdot \mathsf{sh}_n \cdot m^2 \ , \\ N \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m) \quad & \leq \quad N_b \cdot 1 \cdot \mathsf{nb}_{\mathsf{op}_1}(\mathsf{bit}, m) \\ & = \quad N_b \cdot m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m) \quad & \leq \quad N_b \cdot m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(\mathsf{bit}, m) \\ & = \quad N_b \cdot \mathsf{sh}_n \cdot m^2 \ .\end{aligned}$$

– Each multiplication gate requires one multiplication triple. Hence for the multiplication gates we have

$$\begin{aligned}N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{gen}}(x, m) \quad & \leq \quad N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{gen}}(\mathsf{triple}, m) \\ & = \quad N_m \cdot \mathsf{sh}_n \cdot 3m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m) \quad & \leq \quad N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_1}(\mathsf{triple}, m) \\ & = \quad N_m \cdot \mathsf{sh}_n \cdot 2m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m) \quad & \leq \quad N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(\mathsf{triple}, m) \\ & = \quad N_m \cdot \mathsf{sh}_n \cdot m \ .\end{aligned}$$

The randomness seed of λ bits may be generated once for all tuples. Different tuples can be generated in parallel. For simplicity, let μ and κ be the same for generating all types of tuples. Let x_1 be the total number of bits transmitted during the first round, when the initial unverified tuples are shared, and x_2, x_3, x_4 the total number of bits broadcast in the three parallel public openings. The total cost is $(\mathsf{tr}^{\otimes n}_{x_1} \otimes f(\lambda)) \oplus (\mathsf{bc}^{\otimes n}_{x_2} \otimes \mathsf{bc}^{\otimes n}_{x_3} \otimes \mathsf{bc}^{\otimes n}_{x_4})$, where $f(\lambda) := \mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda} \oplus \mathsf{bc}^{\otimes nt}_{\mathsf{sh}_n\cdot\lambda}$ does not

In Π_{verify}, each party works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $\mathsf{comm}[id]$, the function $f(id)$ to verify, and the identifiers $\vec{xid}(id)$ of the inputs on which $f(id)$ should be verified w.r.t. the output identified by id. 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 are stored by the sender and the receiver in a local array sent before they finally get committed to these messages. Π_{verify} uses $\mathcal{F}_{TR}$, $\mathcal{F}_{share}$, and $\mathcal{F}_{pre}$ as subroutines.

• **Initialization:** On input $(\mathsf{init}, f, \vec{xid}, p, p')$, where $\mathsf{Dom}(f) = \mathsf{Dom}(\vec{xid}) = \mathsf{Dom}(p) = \mathsf{Dom}(p')$, store the mappings f, $\vec{xid}$, p, p' for further use. Initialize comm and sent to empty arrays.

Initialize subroutine protocols:

– *Initialize* $\mathcal{F}_{TR}$: For all $id \in \mathsf{Dom}(f)$ s.t $p(id) \neq p'(id)$, define the mappings s, r, f' such that $s(id) \leftarrow p(id)$, $r(id) = f'(id) \leftarrow p'(id)$. For all $i \in [n]$, define an identifier $id' \leftarrow (\mathsf{bc}, i)$ that will be used for broadcast, and $s(id) \leftarrow i$, $r(id) \leftarrow \perp$, $f'(id) \leftarrow \perp$. Send $(\mathsf{init}, s, r, f')$ to $\mathcal{F}_{TR}$.

– *Initialize* $\mathcal{F}_{pre}$: A message $(\mathsf{init}, \bar{m}, \bar{u}, \bar{p})$ is sent to $\mathcal{F}_{pre}$, where $\bar{m}$,$\bar{u}$,$\bar{p}$ 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 $id' \leftarrow (id_i, type, m, u)$ such that $type$ is the type of the tuple, $\bar{m}(id') = m$, $\bar{u}(id') = u$. 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* $\mathbb{Z}_{2^m}$: $(id_i, \mathsf{bit}, m, m)$;
3. *Multiplication in* $\mathbb{Z}_{2^m}$ *:* $(id_i, \mathsf{triple}, m, 1)$;
4. *Extending* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$ *:* $(id_i, \mathsf{bit}, m_y, m_x)$.

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, the tuple generation is optimized by substituting all the identifiers $(id_i, type, m, u_{id_i})$ for the same $type$ and bit width m with a single identifier $id' = (type, m, u)$ for $u = \sum u_{id_i}$. After inducing $\bar{m}, \bar{u}, \bar{p}$ from these new identifiers, each party sends $(\mathsf{init}, \bar{m}, \bar{u}, \bar{p})$ to $\mathcal{F}_{pre}$.

– *Initialize* $\mathcal{F}_{share}$: For commitments of non-random wires, take $\tilde{p}(id) \leftarrow p(id)$, and $\tilde{m}(id) \leftarrow m$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. If $p(id) \neq p'(id)$, generate a new identifier id' and define additionally $\tilde{m}(id') \leftarrow m(id)$, $\tilde{p}(id') \leftarrow p'(id)$. For commitments of random wires, for all $j \in [n]$, for $id_j = (id, j)$, assign $\tilde{m}(id_j) \leftarrow m(id)$, $\tilde{p}(id_j) \leftarrow j$, and for $id'_j = (id', j)$, assign $\tilde{m}(id'_j) \leftarrow m(id)$, $\tilde{p}(id'_j) \leftarrow p(id)$. After doing it for all id, deliver $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{share}$.

Generating precomputed tuples: A message $(type, id')$ is sent to $\mathcal{F}_{pre}$ by each party for each identifier $id' = (type, m, u)$ on which $\mathcal{F}_{pre}$ was initialized. $\mathcal{F}_{pre}$ generates the required tuples and outputs to the parties their shares $\vec{s}^j$. Each party P_j sends a couple of messages $(\mathsf{extcommit}, id^k_v, s^j_k)$ to $\mathcal{F}_{share}$, so that these shares could be used inside $\mathcal{F}_{share}$.

Initialization failure: If $(id, \perp)$ comes from $\mathcal{F}_{pre}$ or $\mathcal{F}_{share}$ for some id, then output $\perp$ to $\mathcal{Z}$.

• **Failed Openings:** At any time during public opening, $\mathcal{F}_{share}$ may output $(id, \perp, \mathcal{K}, \mathsf{deriv}[id])$, where $\mathcal{K}$ is the set of parties that have disagreed with $P_{p(id)}$ about the correctness of the share $\mathsf{comm}^k[id]$. By definition of $\mathcal{F}_{share}$, $|\mathcal{K}| < t$. If $\mathsf{deriv}[id]$ contains any index id' for which P_k for $k \in \mathcal{K}$ has input $(\mathsf{extcommit}, id', x^k)$ before, and that has not been rewritten by $(\mathsf{pcommit}, id', k, x^k)$ afterwards, such message needs to be opened using help of $\mathcal{F}_{pre}$. For any other index id', the shares x^k can be opened directly using $\mathcal{F}_{share}$. In Π_{verify}, the messages $(\mathsf{extcommit}, id', x^i)$ may be input only for such id' for which a value identified by some id'' has been stored in $\mathcal{F}_{pre}$. Hence, if an opening fails, each party P_i behaves as follows for all $k \in \mathcal{K}$:

1. Send $(\mathsf{share\_open}, id'', k, j)$ for corresponding id'' and j to $\mathcal{F}_{pre}$. For any non-external commitment id' of $\mathsf{deriv}[id]$, send $(\mathsf{share\_open}, id', k)$ to $\mathcal{F}_{share}$. If at least one share opening fails, then both $p(id)$ and k should have been corrupted, so $(\mathsf{cheater}, p(id))$ can be output. At most $t - 1$ shares of each value are opened since $|\mathcal{K}| < t$ by definition of $\mathcal{F}_{share}$.
2. If all share openings succeed, after getting back all shares x^k, send $(\mathsf{pcommit}, id', k, x^k)$ for all $k \in \mathcal{K}$, where x^k is the share that corresponds to the identifier id' of $\mathcal{F}_{share}$. Here x^k is treated as a public value, so all parties input the same value x^k.
3. Send (open, id) to $\mathcal{F}_{share}$. All shares of $k \in \mathcal{K}$ are now substituted with public values. $\mathcal{A}$ may abort the opening as far as all $t - 1$ shares of corrupted parties are substituted in this way, and finally $\mathcal{F}_{share}$ outputs either (id, x) or $(\mathsf{cheater}, p(id))$.

• **Cheater detection:** At any time, when $\mathcal{F}_{TR}$ or $\mathcal{F}_{share}$ outputs a message $(\mathsf{cheater}, k)$, output $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. If $(\mathsf{cheater}, p(id))$ comes from $\mathcal{F}_{share}$ during executing (verify, id), then all parties output $(id, 0)$ instead of $(\mathsf{cheater}, p(id))$.

Fig. 23. Real protocol Π_{verify} (initialization,failed openings, cheater detection)

depend on the tuples, and the upper bounds of x_i are

$$\begin{aligned} x_1 &\leq (\mu N_b + \kappa) \cdot \mathsf{sh}_n \cdot m^2 + (\mu N_m + \kappa) \cdot \mathsf{sh}_n \cdot 3m \\ &= \mathsf{sh}_n \cdot m \cdot ((\mu N_b + \kappa) \cdot m + (\mu N_m + \kappa) \cdot 3) \\ &= \mathsf{sh}_n \cdot m \cdot ((\mu (N_b \cdot m + 3N_m) + \kappa(m+3)) \ , \\ x_2 &\leq \kappa \cdot \mathsf{sh}_n \cdot m^2 + \kappa \cdot \mathsf{sh}_n \cdot 3m \\ &= \mathsf{sh}_n \cdot m \cdot \kappa(m+3) \ , \end{aligned}$$

$$\begin{aligned} x_3 &\leq (\mu - 1)(N_b \cdot m + N_m \cdot \mathsf{sh}_n \cdot 2m) \\ &= (\mu - 1)m(N_b + \mathsf{sh}_n \cdot 2N_m) \ , \\ x_4 &\leq (\mu - 1)(N_b \cdot \mathsf{sh}_n \cdot m^2 + N_m \cdot \mathsf{sh}_n \cdot m) \\ &= \mathsf{sh}_n \cdot (\mu - 1)m(N_b \cdot m + N_m) \ . \end{aligned}$$

Substituting x_1, x_2, x_3, x_4 into $(\mathsf{tr}_{x_1}^{\otimes n} \otimes f(\lambda)) \oplus (\mathsf{bc}_{x_2}^{\otimes n} \otimes \mathsf{bc}_{x_3}^{\otimes n} \otimes \mathsf{bc}_{x_4}^{\otimes n})$, we get the final value. $\square$

• **Input Commitment:** On input $(\mathsf{commit\_input}, id, x)$, $P_{p(id)}$ sends (commit, id, x) to $\mathcal{F}_{share}$. On input $(\mathsf{commit\_input}, id)$, each party sends (commit, id) to $\mathcal{F}_{share}$. After getting $(\mathsf{confirmed}, id)$ from $\mathcal{F}_{share}$, $P_{p(id)}$ assigns $\mathsf{comm}[id] \leftarrow x$. Each party outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Message Commitment:**

1. On input $(\mathsf{send\_msg}, id, x)$, $P_{p(id)}$ sends $(\mathsf{transmit}, id, x)$ to $\mathcal{F}_{TR}$. On input $(\mathsf{send\_msg}, id)$, $P_{p'(id)}$ waits for (id, x) from $\mathcal{F}_{TR}$. If the transmission succeeds, both parties assign $\mathsf{sent}[id] \leftarrow x$.
2. On input $(\mathsf{commit\_msg}, id)$, $P_{p(id)}$ sends to $\mathcal{F}_{share}$ the message $(\mathsf{commit}, id, \mathsf{sent}[id])$. Each other party sends $(\mathsf{priv\_open}, id, id')$ to $\mathcal{F}_{share}$. The identifier id' has been defined for this particular id in the initialization phase in such a way that the party committed to $\mathsf{comm}[id']$ of $\mathcal{F}_{share}$ is $P_{p'(id)}$.

If $(id, id', \perp)$ is output by $\mathcal{F}_{share}$, then each party P_j sends (reveal, id) to $\mathcal{F}_{TR}$, and after getting back (id, x), it sends $(\mathsf{pcommit}, id, x)$ and $(\mathsf{pcommit}, id', x)$ to $\mathcal{F}_{share}$. After getting $(\mathsf{confirmed}, id)$ from $\mathcal{F}_{share}$, $P_{p(id)}$ and $P_{p(id')}$ assign $\mathsf{comm}[id] \leftarrow \mathsf{sent}[id]$, and each other party outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, the parties act as follows. Let $\mathcal{T}$ be a set of t parties s.t. $p(id) \notin \mathcal{T}$.

1. Each P_i for $i \in \mathcal{T}$ generates $r_i \overset{\$}{\leftarrow} \mathbb{Z}_{2^{m(id)}}$ and sends $(\mathsf{commit}, id_i, r_i)$ to $\mathcal{F}_{share}$. For $i \neq j$, it sends (commit, id_j) to $\mathcal{F}_{share}$.
2. Upon receiving (id_k, r^i_k) from $\mathcal{F}_{share}$ for all $k \neq p(id)$, each party sends $(\mathsf{priv\_open}, id_k, id'_k)$ for all $k \neq p(id)$ to $\mathcal{F}_{share}$ (this is done to open the value to $P_{p(id)}$).
3. Upon receiving $(id_k, id'_k, (r^j_k)_{j\in[n]})$ for all $k \in \mathcal{T}$ from $\mathcal{F}_{share}$, $P_{p(id)}$ computes $r_k = \mathsf{declassify}(r^j_k)_{j\in[n]}$ for all $k \in \mathcal{T}$, computes $r = \sum_{k\in\mathcal{T}} r_k$, and writes $\mathsf{comm}[id] \leftarrow r$.

If $P_{p(id)}$ complains that it received inconsistent shares r^j_k, each party outputs $\perp$ to $\mathcal{Z}$. Otherwise, it outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Verification:** On input (verify, id), each party P_k checks whether $\mathsf{comm}[i]$ has been defined for all $i \in \vec{xid}(id)$. It decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$. For each f_i, some additional identifiers are defined: $id^{\mathsf{x}k}_i$ for the k-th input, $id^{\mathsf{y}k}_i$ for the k-th output, and $id^{\mathsf{z}k}_i$ 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 are used for intermediate values. Let $id^{type}_{i,k}$ be the index of $\mathcal{F}_{share}$ storing the k-th component of the i-th tuple of type $type$ generated by $\mathcal{F}_{pre}$.

First, $P_{p(id)}$ computes all the intermediate values $\mathsf{comm}[id^{\mathsf{x}k}_i]$ using the function descriptions f_i and the commitments $\mathsf{comm}[i]$ for $i \in \vec{xid}(id)$. Let $\hat{\vec{x}}_i$ denote all values that should be broadcast by $P_{p(id)}$ for verification of f_i, defined as follows:

1-2. *Linear combination, conversion to a smaller ring*: no broadcasts.

3. *Multiplication in* $\mathbb{Z}_{2^m}$: $\hat{\vec{x}}_i \leftarrow [(x_1 - a) \bmod 2^m, (x_2 - b) \bmod 2^m]$, where $a = \mathsf{comm}[id^{\mathsf{triple}}_{i,1}]$, $b = \mathsf{comm}[id^{\mathsf{triple}}_{i,2}]$, $x_1 = \mathsf{comm}[id^{\mathsf{x}1}_i]$, $x_2 = \mathsf{comm}[id^{\mathsf{x}2}_i]$.
4. *Bit decomposition in* $\mathbb{Z}_{2^m}$: $\hat{\vec{x}}_i \leftarrow [c_1, \ldots, c_m]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit $\mathsf{comm}[id^{\mathsf{bit}}_{i,k}]$ is different from the k-th bit of $\mathsf{comm}[id^{\mathsf{x}}_i]$.
5. *Conversion from* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$: Perform bit decomposition of $\mathsf{comm}[id^{\mathsf{x}}_i]$ over $\mathbb{Z}_{2^{n_y}}$, getting n_y bits b_k. Take the first n_x bits. Let $\hat{\vec{x}}_i \leftarrow [c_1, \ldots, c_{n_x}]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit $\mathsf{comm}[id^{\mathsf{bit}}_{i,k}]$ is different from b_k.

$P_{p(id)}$ sends $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id^{type}_i, \hat{\vec{x}}_i)_{i\in[N]})$ to $\mathcal{F}_{TR}$. Upon receiving $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id^{type}_i, \hat{\vec{x}}_i)_{i\in[N]})$, each party writes $\mathsf{pubv}[id^{type}_i] \leftarrow \hat{\vec{x}}_i$. For simplicity of further verifications, we assume that $(id^{\mathsf{bit}}_{i,k} = 1 - id^{\mathsf{bit}}_{i,k})$ 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.
After the verifiers have assigned $\mathsf{pubv}[id^{type}_i]$, they start computing f_i, for all $i \in [N]$. The basic operations f_i are computed by sending lc and trunc to $\mathcal{F}_{share}$. The outputs of f_i are stored in $\mathcal{F}_{share}$ under identifiers id^{y}_i (used also as $id^{\mathsf{x}}_{i'}$ for computing the next basic operations $f_{i'}$ for $i' > i$), and the alleged zeroes are stored in $\mathcal{F}_{share}$ under identifiers id^{z}_i.

1. *Linear combination* $[c_0, \ldots, c_l]$: Send $(id^y_i = c_0 + \sum_{k\in[l]} c_k \cdot id^{x_k}_i)$ to $\mathcal{F}_{share}$.
2. *Conversion to a smaller ring* $\mathbb{Z}_{2^m}$: Send $(id^{\mathsf{y}}_i = id^{\mathsf{x}}_i \bmod 2^m)$ to $\mathcal{F}_{share}$.
3. *Multiplication in* $\mathbb{Z}_{2^m}$: Let $(id^a, id^b, id^c) = (id^{\mathsf{triple}}_{i,k})_{k\in[3]}$, and $[\hat{x}_1, \hat{x}_2] = \mathsf{pubv}[id^{\mathsf{triple}}_i]$. Send to $\mathcal{F}_{share}$:
   - $id^{\mathsf{y}}_i = \hat{x}_1 \cdot id^b + \hat{x}_2 \cdot id^a + id^c + \hat{x}_1 \cdot \hat{x}_2$;
   - $id^{\mathsf{z}1}_i = \hat{x}_1 + id^a - id^{\mathsf{x}1}_i$;
   - $id^{\mathsf{z}2}_i = \hat{x}_2 + id^b - id^{\mathsf{x}2}_i$.
4. *Bit decomposition in* $\mathbb{Z}_{2^m}$: Let $[id^{b_1}, \ldots, id^{b_m}] = (id^{\mathsf{bit}}_{i,k})_{k\in[m]}$. Send $(id^{\mathsf{z}}_i = id^{\mathsf{x}}_i - \sum_{k=1}^{m} 2^{k-1} \cdot id^{b_k})$ to $\mathcal{F}_{share}$.
5. *Conversion from* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$: Let $[id^{b_1}, \ldots, id^{b_{m_x}}] = (id^{\mathsf{bit}}_{i,k})_{k\in[m_x]}$. Send to $\mathcal{F}_{share}$
   - $id^{\mathsf{y}}_i = \sum_{k=1}^{m_x} 2^{k-1} \cdot id^{b_k}$;
   - $id^{\mathsf{w}}_i = id^{\mathsf{y}}_i \bmod m_x$;
   - $id^{\mathsf{z}}_i = id^{\mathsf{x}}_i - id^{\mathsf{w}}_i$.

Send also $(id^{\mathsf{z}k}_{N+1} = id^{\mathsf{y}k}_N - id)$ to $\mathcal{F}_{share}$, to verify the final output against $\mathsf{comm}[id]$.
After all f_i have been processed, for each alleged zero $id^{\mathsf{z}k}_i$, send $(\mathsf{open}, id^{\mathsf{z}k}_i)$ to $\mathcal{F}_{share}$. Upon receiving all $(id^{\mathsf{z}k}_i, z_{ik})$ from $\mathcal{F}_{share}$, if $z_{ik} = 0$ for all i, k, then output $(id, 1)$. Otherwise, output $(id, 0)$.

Fig. 24. Real protocol Π_{verify} (commitments, verification)

Observation 7. *From the description of the commitment functions of Π_{verify}, we count the number of $\mathcal{F}_{TR}$ and $\mathcal{F}_{share}$ calls that it makes. They are given in Table 12. The cost of broadcasting the complaint* (bad, k) *is omitted.*

Observation 8. *By counting the number of the broadcast hint bits an the alleged zero bits for each basic operation, we get the numbers given in Table. 13. Note that the hint bits c_i broadcast for each bit decomposition do not have to be committed in $\mathbb{Z}_{2^m}$, and each such bit is broadcast as a 1-bit value.*

Lemma 9 (cost of the broadcasts of $\mathcal{F}_{verify}$)**.** *Let f be a function that is going to be verified. Let f consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}\}$. Let 2^m be the size of the largest used ring. The total cost of the broadcast phase of Π_{verify} is upper bounded by $\mathsf{bc}_{N \cdot 2m}$.*

Proof. All the bits are broadcast in parallel using $\mathcal{F}_{TR}$. We use Table 13 to count the number of bits for each operation. We take the upper bound $2m$ on broadcast bits per operation, which comes from multiplication. Differently from the initialization phase of Π_{verify}, the costs are similar for distinct types of basic operations, and they are all $O(m)$. □

Lemma 10 (cost of the final verification of Π_{verify})**.** *Let all the functions f to be verified consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}\}$. Let M_y be the total number of bits output by f. Let M_x, M_r, M_c, M_{pre} be the total number of bits in the committed input, randomness, communicated elements, and precomputed tuples respectively. Let 2^m be the size of the largest used ring. The cost of the verification phase of Π_{verify} is upper bounded by:*

- $\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot (N \cdot 2m + M_y)}$ *if all openings* (open, id) *succeed,*
- $\mathsf{rev}^{\otimes (t-1)}_{\mathsf{sh}_n \cdot (M_x + M_r + M_c + M_{pre} + M_y)}$ *if some* (open, id) *outputs* $(id, \bot)$.

Proof. Taking into account the costs of different operations of Π_{share} given in Obs. 3, the functionalities lc and trunc do not take any communication. Hence the cost of verifying basic operations comes only from the hint broadcast and the alleged zero check.

- Assume that (open, id) succeeds for all alleged zeroes. It has cost $\mathsf{bc}^{\otimes n}_M$ for an M-bit value. From Table. 13, we see that the largest number of alleged zero checks per operation is $2m$ that comes from multiplication. 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 (open, id) returns $(id, \bot)$ for some alleged zero identifier id. In this case the shares held by parties of $\mathcal{K}$ should be revealed, which may include all the initial inputs from which the alleged zeroes and the outputs were computed, and also the shares of the final outputs. By definition of $\mathcal{F}_{share}$, $|\mathcal{K}| < t$, so up to $t-1$ shares are revealed. □

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 $\mathcal{F}_{TR}$-$\mathcal{F}_{share}$-$\mathcal{F}_{pre}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{verify}$ described in Fig. 25. It runs a local copy of Π_{verify}, together with local copies of $\mathcal{F}_{TR}$, $\mathcal{F}_{share}$, $\mathcal{F}_{pre}$.

Preprocessing. For the generation of precomputed tuples, $\mathcal{S}$ simulates $\mathcal{F}_{pre}$. If the generation succeeds, $\mathcal{A}$ assumes that the triples generated by $\mathcal{F}_{pre}$ are all copied to $\mathcal{F}_{share}$, since at least the honest parties have input $(\mathsf{extcommit}, id, x^k)$ to $\mathcal{F}_{share}$ for the corresponding identifiers id and the shares x^k that they received from $\mathcal{F}_{pre}$. If it fails, then the parties should output $\bot$ in the real execution. In this case, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{verify}$, so that $\bot$ is output to $\mathcal{Z}$ by $\mathcal{F}_{verify}$, as it would be expected from a real protocol execution.

Input and message commitments. For input and message commitments, $\mathcal{S}$ simulates $\mathcal{F}_{share}$ and $\mathcal{F}_{TR}$, where the inputs of corrupted parties are provided by $\mathcal{A}$, and the inputs of honest parties that should be delivered to a corrupted party are given to $\mathcal{S}$ by $\mathcal{F}_{verify}$. Up to $t-1$ shares need to be simulated to $\mathcal{A}$, and these may be sampled by $\mathcal{S}$ from a uniform distribution.

For input commitments, if $p(id) \in \mathcal{C}$, the commitments may fail. In this case $\mathcal{S}$ delivers to $\mathcal{F}_{verify}$ the message $(\mathsf{cheater}, p(id))$.

For message commitments, $\mathcal{S}$ simulates sending (commit, id, x), (commit, id), and $(\mathsf{priv\_open}, id, id')$ to $\mathcal{F}_{share}$. At this point, it either $p(id) \in \mathcal{C}$ or $p'(id) \in \mathcal{C}$, then $\mathcal{S}$ has already simulated x before to both $\mathcal{A}$ and to $\mathcal{F}_{share}$, so it uses the same x.

If either $(id, id', \bot)$, $(\mathsf{cheater}, p(id))$, or $(\mathsf{cheater}, p'(id))$ is output by $\mathcal{F}_{share}$, or $\mathcal{A}$ has chosen that $\mathcal{F}_{share}$ outputs to $P_{p(id')}$ some $x' \neq x$, then $\mathcal{S}$ simulates work of $\mathcal{F}_{TR}$ on input (reveal, id). The latter results in opening (id, x) to $\mathcal{A}$, where x is the value that was actually transmitted, and since $\mathcal{F}_{share}$ may fail only if $p(id) \in \mathcal{C}$ or $p'(id) \in \mathcal{C}$, $\mathcal{S}$ takes the same x that it has already used in the simulation before.

• **Initialization:** On input $(\mathsf{init}, f, \vec{xid}, p, p')$, from $\mathcal{F}_{verify}$, $\mathcal{S}$ initializes its local copies of $\mathcal{F}_{pre}$, $\mathcal{F}_{share}$, $\mathcal{F}_{TR}$ as parties in Π_{verify} do. Then $\mathcal{S}$ simulates running $\mathcal{F}_{pre}$ to generate tuples. This does not require any knowledge from $\mathcal{F}_{verify}$. If the generation has not failed, then $\mathcal{A}$ expects that all (valid) tuples are copied to $\mathcal{F}_{share}$. If the generation fails, then $\mathcal{S}$ delivers (stop) to $\mathcal{F}_{verify}$, so that $\perp$ is output in both the real and the simulated executions.

• **Input Commitment:** On inputs $(\mathsf{commit\_input}, id, x)$ and $(\mathsf{commit\_input}, id)$, $\mathcal{S}$ simulates sending (commit, id, x) and (commit, id) to $\mathcal{F}_{share}$. If $p(id) \in \mathcal{C}$, then $\mathcal{A}$ chooses $(x^k)_{k\in[n]}$ for $\mathcal{F}_{share}$. $\mathcal{S}$ computes $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$ and delivers x to $\mathcal{F}_{verify}$.

• **Message Commitment:** On inputs $(\mathsf{send\_msg}, id, x)$ and $(\mathsf{send\_msg}, id)$, $\mathcal{S}$ simulates sending $(\mathsf{transmit}, id, x)$ to $\mathcal{F}_{TR}$. On input $(\mathsf{commit\_msg}, id)$, $\mathcal{S}$ simulates sending (commit, id, x), (commit, id), and $(\mathsf{priv\_open}, id, id')$ to $\mathcal{F}_{share}$ by the corresponding parties. If $p(id) \in \mathcal{C}$, then $\mathcal{A}$ chooses $(x^k)_{k\in[n]}$, and $\mathcal{S}$ delivers $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$ to $\mathcal{F}_{verify}$. As a side-effect, $\mathcal{F}_{share}$ requires to deliver up to $t-1$ shares of x of $p(id) \notin \mathcal{C}$ to $\mathcal{A}$. $\mathcal{S}$ samples them from uniform distribution.

If $(id, id', \perp)$, $(\mathsf{cheater}, p(id))$, or $(\mathsf{cheater}, p'(id))$ should be output by $\mathcal{F}_{share}$, then $\mathcal{S}$ simulates sending (reveal, id) to $\mathcal{F}_{TR}$. It then writes $\mathsf{comm}[id] \leftarrow x$ in its local copy of $\mathcal{F}_{share}$, where x is the value that was initially transmitted.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, $\mathcal{S}$ needs to simulate sending $(\mathsf{commit}, id_j, r_j)$ and $(\mathsf{priv\_open}, id_j, id'_j)$ to $\mathcal{F}_{share}$.

- For $p(id) \in \mathcal{C}$, $\mathcal{S}$ needs to enforce commitment of the particular randomness r that it receives from $\mathcal{F}_{verify}$. The commitments r_j for $j \notin \mathcal{C}$ are simulated in exactly the same way as by $\mathcal{S}_{coins}$, and we now repeat it here. As far the opening has not started, $\mathcal{A}$ does not expect to see the values r_j generated by $j \notin \mathcal{C}$. Since opening may start only on all honest parties' agreement, $\mathcal{S}$ is free to wait with their generation until $\mathcal{A}$ commits to r_j for all $j \in \mathcal{C}$. After that, $\mathcal{S}$ generates r_j for $j \notin \mathcal{C}$: up to the last share, it samples $r_j \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, and then computes the last share as $r_j = r - \sum_{k=1,k\neq j}^{n} r_k$. This is always possible if at least t parties contribute r_j.
- For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ does not get r from $\mathcal{F}_{verify}$, but it may generate arbitrary shares $r_j \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$ since r will not have to be opened to $\mathcal{A}$ anyway.

After (id_j, r_j^k) has been output to each party P_k, $\mathcal{A}$ waits for $(\mathsf{priv\_open}, id_j, id'_j)$ for all $j \in [n] \setminus \{p(id)\}$. $\mathcal{S}$ uses the previously generated r_j to simulate $\mathsf{priv\_open}$. If $\mathsf{priv\_open}$ fails, then the parties should output $\perp$ in the real execution. In this case, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{verify}$, so that $\perp$ is output to $\mathcal{Z}$, as it would expect from a real protocol execution.

• **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 $\mathsf{comm}[id_i^{\mathsf{x}_k}]$ and $\mathsf{comm}[id_i^{\mathsf{y}_k}]$, and broadcasts the values $\hat{\vec{x}}$ chosen by $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, broadcasting $\hat{\vec{x}}$ is to be simulated by $\mathcal{S}$ as follows (we use case distinction on types of precomputed tuples causing the broadcast):

1. *Bit decomposition of x in $\mathbb{Z}_{2^m}$*: Need to broadcast $\hat{\vec{x}}_i = [c_1, \ldots, c_m]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit b_k is different from the k-th bit of x. Generate $c_k \xleftarrow{\$} \{0,1\}$.
2. *Multiplication of x_1 and x_2 in $\mathbb{Z}_{2^m}$*: Need to broadcast $\hat{\vec{x}}_i = [(x_1 - a) \bmod 2^m, (x_2 - b) \bmod 2^m]$ for the multiplication triple (a, b, c). Generate $\hat{\vec{x}}_i \xleftarrow{\$} \mathbb{Z}_{2^m}^2$.

$\mathcal{S}$ simulates $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id_i^{type}, \hat{\vec{x}}_i)_{i\in[N]})$ using $\mathcal{F}_{TR}$. If the broadcast succeeds and no $(\mathsf{cheater}, p(id))$ should be output, $\mathcal{S}$ writes $\mathsf{pubv}[id_i^{type}] \leftarrow \hat{\vec{x}}_i$ for all honest parties. For the trusted bits, it simulates sending the corresponding messages $(id_{i,k}^{\mathsf{bit}} = 1 - id_{i,k}^{\mathsf{bit}})$ to $\mathcal{F}_{share}$ for $c_k = 1$, as the honest parties do.

The further computation depends on f_i, and S just simulates sending to $\mathcal{F}_{share}$ the same messages that the honest parties send (linear combinations and truncations). These operations do not involve any interaction between parties, so nothing needs to be simulated to $\mathcal{A}$.

In the end, $\mathcal{S}$ needs to simulate opening to each party the alleged zero vector $\vec{z}$. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ holds all the values needed to compute $\vec{z}$, that have already been delivered to $\mathcal{F}_{verify}$ during the simulations of commitments. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ obtains the difference $z = f(\vec{x}) - y$ from $\mathcal{F}_{verify}$. It takes $\vec{z}_{N+1} \leftarrow [z]$, and $\vec{z}_i \leftarrow \vec{0}$ for all other $i \in \{1, \ldots, N\}$.

• **Failed Openings:** At any time when a public opening fails, $\mathcal{S}$ simulates $(id, \perp, \mathcal{K}, \mathsf{deriv}[id])$ being output to all parties, where the set $\mathcal{K}$ comes from the simulation of $\mathcal{F}_{share}$. $\mathcal{S}$ needs to simulate the share opening, which is done either using $\mathcal{F}_{share}$ or $\mathcal{F}_{pre}$ (depending on whether the corresponding leaf of $\mathsf{deriv}[id]$ is a local or an external commitment). $\mathcal{S}$ simulates using $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$ to open the shares x^k of parties belonging to the set $\mathcal{K}$.

- If $p(id) \in \mathcal{C}$, then $\mathcal{A}$ is allowed to open any shares x^k for $k \in \mathcal{C}$ using $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$. For $k \notin \mathcal{C}$, $\mathcal{S}$ simulates opening the same x^k that have been simulated to $\mathcal{A}$ during the commitments of $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$.
- If $p(id) \notin \mathcal{C}$, we have $\mathcal{K} \subseteq \mathcal{C}$ by definition of $\mathcal{F}_{share}$. Hence $k \notin \mathcal{C}$ is not possible, so opening x^k should be simulated only for $k \in \mathcal{C}$, and $\mathcal{S}$ takes the same x^k that has been used in the simulation of $\mathcal{F}_{share}$ or $\mathcal{F}_{pre}$.

• **Cheater detection:** At any time, when $\mathcal{F}_{TR}$ or $\mathcal{F}_{share}$ should output a message $(\mathsf{cheater}, k)$, $\mathcal{S}$ outputs $(\mathsf{cheater}, k)$ to $\mathcal{F}_{verify}$.

Fig. 25. The simulator $\mathcal{S}_{verify}$

Table 12. Number of $\mathcal{F}_{share}$ and $\mathcal{F}_{TR}$ operations needed for committing M values of N bits each in $\mathcal{F}_{verify}$

commitment	call	# calls	total $\mathcal{F}_{TR}$ cost
commit_input	commit	1	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nM}$
send_msg	transmit	1	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes M}$
commit_msg (optimistic)	commit	1	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nM}$
	priv_open	1	$\mathsf{fwd}_{\mathsf{sh}_n \cdot N}^{\otimes nM} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nM}$
commit_msg (if $P_{p(id)}$ and $P_{p'(id)}$ disagree)	reveal	1	$\mathsf{rev}_{\mathsf{sh}_n \cdot N}^{\otimes M}$
commit_rnd	commit	t	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes ntM}$
	priv_open	t	$\mathsf{fwd}_{\mathsf{sh}_n \cdot N}^{\otimes ntM} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes ntM}$

Table 13. Number of bits for verifying each operation in $\mathcal{F}_{verify}$

operation	# hint bits	# alleged zero bits
Linear combination	0	0
Conv. to a smaller ring	0	0
Bit decomposition in $\mathbb{Z}_{2^m}$	m	m
Multiplication in $\mathbb{Z}_{2^m}$:	$2m$	$2m$
Extending $\mathbb{Z}_{2^{m_x}}$ to $\mathbb{Z}_{2^{m_y}}$	m_x	m_x

For the randomness commitments, $\mathcal{S}$ gets r from $\mathcal{F}_{verify}$. $\mathcal{S}$ needs to simulate r_j generated by $j \notin \mathcal{C}$ in such a way that r will be finally committed to $\mathcal{F}_{share}$ in the simulation. The generation of appropriate r_j is done in the same settings as for $\mathcal{S}_{coins}$, so we refer to the proof of Lemma 5 here. As the result, for $p(id) \in \mathcal{C}$, $\mathcal{A}$ believes that it should be $\mathsf{comm}[id] = r$ in the inner state of $\mathcal{F}_{share}$ in the real protocol. For $p(id) \notin \mathcal{C}$, the value r does not have to be simulated to $\mathcal{A}$. All the commitments of r_j are simulated using the local copy of $\mathcal{F}_{share}$. After that, $\mathcal{S}$ needs to simulate the private openings of the shares. If priv_open fails, then the parties should output $\perp$ in the real execution. In this case, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{verify}$, so that $\perp$ is output to $\mathcal{Z}$ by $\mathcal{F}_{verify}$, as expected from a real protocol execution.

For all commitments, the values of $\mathsf{comm}[id]$ inside $\mathcal{F}_{verify}$ are consistent with the view of $\mathcal{A}$ of $\mathsf{comm}[id]$.

Verification. When the verification starts, $\mathcal{S}$ needs to simulate the broadcast. It needs to generate the values of the honest provers itself. All these values are some private values hidden by a random mask (each tuple is used only once), and hence are distributed uniformly:

1. *Bit decomposition of x in $\mathbb{Z}_{2^m}$*: 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 $\mathbb{Z}_{2^m}$*: Since the entries a and b of the triple (a, b, c) are distributed uniformly in $\mathbb{Z}_{2^m}$, so are the values $(x_1 - a) \bmod 2^m$ and $(x_2 - b) \bmod 2^m$.

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

We need to show that the verification either succeeds in both the real and the simulated execution, or it fails in both. The inputs [messages] of $p(id) \notin \mathcal{C}$, the randomness chosen by $\mathcal{F}_{verify}$, and the inputs [messages] of $p(id) \in \mathcal{C}$ chosen by $\mathcal{A}$ are all stored in $\mathcal{F}_{share}$. In addition, the precomputed tuples are also stored in the same $\mathcal{F}_{share}$ since the honest parties have transferred there the shares they obtained from $\mathcal{F}_{pre}$. Now $\mathcal{F}_{share}$ may be used as a black box, doing computation on all these commitments. It remains to prove that, if all these values are committed properly (it is ensured by $\mathcal{F}_{share}$ on the condition that $(\mathsf{cheater}, p(id))$ is not output for the prover $P_{p(id)}$), then Π_{verify} does verify the computation of $f(id)$ on input (verify, id).

It easy to see that, if $\vec{z}_i = \vec{0}$ 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 $\hat{\vec{x}}_i$ has been computed correctly for f_i. The details of verifying each basic function are analogous to the precomputed tuple generation proof of Lemma 7, so we do not repeat the proof here. If all f_i have been computed correctly, then so is the composition of f.

Conversely, if $\vec{z} \neq \vec{0}$ in Π_{verify}, it does not immediately imply that $f(\vec{x}) - y = 0$, since the problem might be in the values broadcast by the prover. The parties of Π_{verify} output $(id, 0)$ in this case. Here we use the fact that $\mathcal{F}_{verify}$ also outputs $(id, 0)$ to the parties instead of $(\mathsf{cheater}, p(id))$ during the execution of (verify, id).

Failed openings. In the previous steps, we always assumed for simplicity that the opening always succeeds. If $(\mathsf{cheater}, p(id))$ is output instead, then verification does not need to proceed further, since the guilt of $P_{p(id)}$ is already proven, so it is also not a problem. However, the opening may fail without identifying the cheater. In this case, all parties get from $\mathcal{F}_{share}$ the message $(id, \perp, \mathcal{K}, \mathsf{deriv}[id])$, where $\mathcal{K} < t$. $\mathcal{S}$ needs to simulate $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$ opening the shares x^k of $\mathcal{K}$.

- If $p(id) \in \mathcal{C}$, then $\mathcal{A}$ is allowed to open any shares x^k for $k \in \mathcal{C}$ using $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$. However, for $k \notin \mathcal{C}$, $\mathcal{A}$ cannot prevent opening of x^k that has been actually given to an honest P_k before, and hence $\mathcal{S}$ simulates opening the same x^k that have been simulated to $\mathcal{A}$ during the commitments of $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$. Since $\mathcal{A}$ is not able to modify even a single share of an honest party, the value of $\mathsf{comm}[id]$, will not change inside $\mathcal{F}_{share}$.
- If $p(id) \notin \mathcal{C}$, we have $\mathcal{K} \subseteq \mathcal{C}$ by definition of $\mathcal{F}_{share}$ (an honest party never complains against another honest party). Hence $k \in \mathcal{C}$ is not possible, so opening x^k should be simulated only for $k \in \mathcal{C}$, and $\mathcal{S}$ takes the same x^k that have been simulated to $\mathcal{A}$ during the commitments of $\mathcal{F}_{share}$ and $\mathcal{F}_{pre}$. By definition of these functionalities, $\mathcal{A}$ cannot force opening of these x^k to fail or substitute them with other values. Therefore, the values of leaves $\mathsf{comm}[id']$ of $\mathsf{deriv}[id]$, and hence also the value of $\mathsf{comm}[id]$, will not change inside $\mathcal{F}_{share}$.

In both cases, there is no way for $\mathcal{A}$ to modify the value of $\mathsf{comm}[id]$ that is stored inside $\mathcal{F}_{share}$, and hence a failed opening does not affect any further opening. On the next input (open, id), complaints of $\mathcal{K}$ will no longer be taken into account, and if $p(id) \notin \mathcal{C}$, then the value $\mathsf{comm}[id]$ will finally be opened. For $p(id) \in \mathcal{C}$, $\mathcal{A}$ may choose to output $(\mathsf{cheater}, p(id))$ instead.

Summary. The entire simulated view of $\mathcal{A}$, including the preprocessing, all commitments, and the verification, is consistent with the inputs and outputs of $\mathcal{F}_{verify}$. Hence the view of $\mathcal{A}$ is the same as it would be in a real protocol execution where the parties have same inputs and outputs as $\mathcal{F}_{verify}$ does, and no $\mathcal{Z}$ can distinguish the real execution from the simulated one. □

C.3 The Main Protocol for Verifiable SMC

The protocol Π_{vmpc} implementing $\mathcal{F}_{vmpc}$ is given in Fig. 26. The protocol is built on top if the functionality $\mathcal{F}_{verify}$ of Sec. C.2, used to verify the computation of each output of each round, with respect to the committed inputs, messages, and randomness. The execution of both $\mathcal{F}_{vmpc}$ and Π_{vmpc} takes place only if their initialization (generation of randomness and precomputed tuples) succeeds. During execution of Π_{vmpc}, cheating is detected as follows.

- If a party P_k refuses to properly commit or transmit messages, $\mathcal{F}_{verify}$ outputs $(\mathsf{cheater}, k)$ on inputs $(\mathsf{commit\_input}, id)$, $(\mathsf{send\_msg}, id)$, $(\mathsf{commit\_msg}, id)$.
- If a party P_k has cheated in the execution phase, $\mathcal{F}_{verify}$ outputs $(id, 0)$ on input (verify, id), where id identifies some output of P_k.
- If a party cheats during the verification, $\mathcal{F}_{verify}$ outputs $(\mathsf{cheater}, k)$ on input (verify, id).

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. 27. The simulator runs a local copy of Π_{vmpc}, together with a local copy of $\mathcal{F}_{verify}$.

Preprocessing phase. The preprocessing phase of $\mathcal{F}_{vmpc}$ and Π_{vmpc} corresponds to their initialization and the randomness generation. Getting the randomness $\vec{r}_i$ for $i \in \mathcal{C}$ from $\mathcal{F}_{vmpc}$, $\mathcal{S}$ simulates initialization of $\mathcal{F}_{verify}$ and the randomness commitment. If $\mathcal{F}_{verify}$ outputs $\perp$, then $\mathcal{S}$ delivers (stop) to $\mathcal{F}_{vmpc}$, so that both the real and the simulated executions abort.

Execution phase. For the private input commitments $(\mathsf{commit\_input})$, $\mathcal{S}$ only needs inputs of corrupted parties for simulation of $\mathcal{F}_{verify}$. All of them are given to $\mathcal{S}$ by $\mathcal{A}$, and $\mathcal{S}$ delivers them to $\mathcal{F}_{vmpc}$, so that it would use the same inputs. If $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{verify}$ during input commitments, then $\mathcal{S}$ sends (stop) to $\mathcal{F}_{vmpc}$, so that the execution does not proceed neither in $\mathcal{F}_{vmpc}$ nor Π_{vmpc} (the party that refused to commit could be blamed by all parties in this case). During the execution phase, $\mathcal{S}$ needs to simulate to $\mathcal{A}$ the messages $\vec{m}^{\ell}_{ij}$ that are computed by the honest parties P_i for corrupted parties P_j $(\mathsf{send\_msg})$. It gets all such messages from $\mathcal{F}_{vmpc}$. On the other hand, the messages $\vec{m}^{\ell}_{ij}$ that are computed by corrupted P_i are chosen by $\mathcal{A}$, and $\mathcal{S}$

In Π_{verify}, each party P_i maintains a local array mlc_i of length n, into which it marks the parties that have been detected in violating the protocol rules. Initially, $mlc_i[k] = 0$ for all $k \in [n]$. If P_k has been detected in cheating, P_i writes $mlc_i[k] = 1$. Π_{vmpc} uses $\mathcal{F}_{verify}$ as a subroutine.

• **In the beginning**, Each party P_i gets the message $(\mathsf{circuits}, (C^\ell_{ij})^{n,n,r}_{i,j,\ell=1,1,1})$ 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 $id \leftarrow (i, \mathsf{r}, k)$, let $\mathbb{Z}_{2^m}$ be the ring in which the wire is defined. Define $f(id) \leftarrow \mathsf{id}_{\mathbb{Z}_{2^m}}$, $\vec{xid}(id) \leftarrow [id]$, $p(id) = p'(id) = i$.
   – For each output wire $id \leftarrow (i, j, \ell, k)$, define $f(id)$ as a function consisting of basic operations of Sec. 3, computing the k-th coordinate of $\vec{m}^\ell_{ij} \leftarrow C^\ell_{ij}(\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell-1}_{ni})$ (this is always possible since every gate of C^ℓ_{ij} is by definition some basic operation), $\vec{xid}(id)$ the vector of all the identifiers of $\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell-1}_{ni}$ that are actually used by C^ℓ_{ij}, $p(id) = i$, $p'(id) = j$.

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

• **For each round** $\ell \in [r]$, P_i computes $\vec{m}^\ell_{ij} = C^\ell_{ij}(\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell-1}_{ni})$ for all $j \in [n]$, and sends $(\mathsf{send\_msg}, (i, j, \ell, k), m^\ell_{ijk})$ to $\mathcal{F}_{verify}$ for all $k \in [|\vec{m}^\ell_{ij}|]$.

• **After r rounds**, each party P_i outputs $(\mathsf{output}, \vec{m}^r_{1i}, \ldots, \vec{m}^r_{ni})$ 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 $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{verify}$, each 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 party sends $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n]$, $\ell \in [r']$, $k \in [n^\ell_{ij}]$. If $(\mathsf{cheater}, i)$ comes from $\mathcal{F}_{verify}$, then each party P_j writes $mlc_j[i] \leftarrow 1$, and the verification of P_i is treated as failed, without even running (verify, id).
2. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, each party sends (verify, id) to $\mathcal{F}_{verify}$, getting the answer (id, b) from $\mathcal{F}_{verify}$. If $b = 1$, each party P_j writes $mlc_j[i] \leftarrow 0$. Otherwise, it writes $mlc_j[i] \leftarrow 1$.

• **Finally**, each party 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. 26. The protocol Π_{vmpc} for verifiable computations

delivers them to $\mathcal{F}_{vmpc}$ to maintain consistency of all messages in the real and the simulated execution.

Verification phase. At the beginning of the verification phase, $\mathcal{S}$ simulates commitments to the messages ($\mathsf{commit\_msg}$) that have been transmitted before. If both the sender and the receiver of the transmitted message are corrupted, then $\mathcal{F}_{verify}$ allows $\mathcal{A}$ to choose m to commit. In this case, $\mathcal{S}$ delivers m to $\mathcal{F}_{vmpc}$, so that it would take into account exactly the same messages as in the simulated real execution.

After all messages have been committed, or $(\mathsf{cheater}, k)$ output for $k \in \mathcal{C}$ for parties that refuse to commit, $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs (verify, id), for all circuit output identifiers id. For this, it needs to simulate the final decision of $\mathcal{F}_{verify}$, and also its side-effect $f(\vec{x}) - y$. 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 $(\mathsf{cheater}, p(id))$ has been output to each honest party (in this case, verification is not executed for $p(id)$ at all), 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 $f(\vec{x}) - y$ should be 0 for $p(id) \in \mathcal{C}$, and so it is easy to simulate. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ computes $f(x) - y$ directly from x and y that have been committed to $\mathcal{F}_{verify}$ before.

It remains to prove that $\mathcal{F}_{vmpc}$ finally outputs exactly the same values as the parties in Π_{vmpc} would. Since $\mathcal{S}$ has so far simulated to $\mathcal{A}$ the messages that were chosen by $\mathcal{F}_{vmpc}$, and it has delivered to $\mathcal{F}_{vmpc}$ all messages that have been chosen by $\mathcal{A}$, it would prove that the simulated execution is indistinguishable from a real protocol. $\mathcal{F}_{vmpc}$ has two kinds of outputs:

1. *The computation output* $(\mathsf{output}, \vec{m}^r_{1i}, \ldots, \vec{m}^r_{ni})$. Let ℓ be any round. We prove by induction that each message $\vec{m}^\ell_{ij}$, seen by the adversary, is consistent with $\mathcal{F}_{vmpc}$'s internal state.

The simulator $\mathcal{S}$ maintains the commitments $\mathsf{comm}[id]$ of the identifiers id denoting the circuit wires whose values are known to the corrupted parties.

• **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 $\mathcal{F}_{verify}$. For $i \in \mathcal{C}$, it adjusts the randomness generator of $\mathcal{F}_{verify}$ in such a way that it generates exactly the same randomness $\vec{r}_i$ for P_i as chosen by $\mathcal{F}_{vmpc}$.
2. *Randomness generation:* $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_rnd}, id)$ for each input wire $id \leftarrow (i, \mathsf{r}, k)$. For all $i \in \mathcal{C}$, the committed randomness r is taken according to $\vec{r}_i$ that has been chosen by $\mathcal{F}_{vmpc}$. $\mathcal{S}$ writes $\mathsf{comm}[id] \leftarrow r$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_input}, id, x_{ik})$ and $(\mathsf{commit\_input}, id)$, where $\vec{x}_i$ is the vector of inputs of the party P_i. For $i \in \mathcal{C}$, the vector $\vec{x}^*_i$ is chosen by $\mathcal{A}$. $\mathcal{S}$ delivers this $\vec{x}^*_i$ to $\mathcal{F}_{vmpc}$, and writes $\mathsf{comm}[id] \leftarrow x^*_{ik}$ for all $id \leftarrow (i, \mathsf{x}, k)$.

In any of these points, if $\perp$ or $(\mathsf{cheater}, k)$ is output by $\mathcal{F}_{verify}$, send (stop) to $\mathcal{F}_{vmpc}$ to stop it. The protocol execution does not start in this case.

• **For each round** $\ell \in [r]$, $\mathcal{S}$ needs to simulate computing the messages $\vec{m}^{\ell}_{ij} = C^{\ell}_{ij}(\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell-1}_{ni})$ for all $j \in \mathcal{C}$. If $i \in \mathcal{C}$, then the message $\vec{m}^{*\ell}_{ij}$ is generated by the adversary, and $\mathcal{S}$ delivers it to $\mathcal{F}_{vmpc}$. If $j \in \mathcal{C}$, then the message $\vec{m}^{\ell}_{ij}$ comes from $\mathcal{F}_{vmpc}$, and $\mathcal{S}$ delivers it to $\mathcal{A}$. In all cases, $\mathcal{S}$ simulates sending $(\mathsf{send\_msg}, (i, j, \ell, k), m^{\ell}_{ijk})$ to $\mathcal{F}_{verify}$ for each entry m^{ℓ}_{ijk} of $\vec{m}^{\ell}_{ij}$.

• **After r rounds**, each party P_i should output $(\mathsf{output}, \vec{m}^r_{1i}, \ldots, \vec{m}^r_{ni})$ 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 $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{verify}$, $\mathcal{S}$ writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. In this case the outputs are not sent to $\mathcal{Z}$. $\mathcal{S}$ defines $\mathcal{B}_0 = \{k \mid (\mathsf{cheater}, k) \text{ has been output}\}$, 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 $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n]$, $\ell \in [r']$, $k \in [n^{\ell}_{ij}]$. If either $i \in \mathcal{C}$ or $j \in \mathcal{C}$, it writes $\mathsf{comm}[(i, j, \ell, k)] \leftarrow m^{\ell}_{ijk}$. If both $i, j \in \mathcal{C}$, then $\mathcal{A}$ chooses $m^{*\ell}_{ijk}$, and $\mathcal{S}$ writes $\mathsf{comm}[(i, j, \ell, k)] \leftarrow m^{*\ell}_{ijk}$
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 $\mathcal{F}_{verify}$.
   - Let $k \in \mathcal{C}$. If $f'(\mathsf{comm}[i]_{i \in \vec{x}\vec{id}}) \neq \mathsf{comm}[id]$ for $f' := f(id)$, then $\mathcal{S}$ simulates $\mathcal{F}_{verify}$ outputting $(id, 0)$, and writes $mlc_i[k] \leftarrow 1$ for all $i \notin \mathcal{C}$. Otherwise, it simulates $\mathcal{F}_{verify}$ outputting $(id, 1)$, and writes $mlc_i[k] \leftarrow 0$. The side-effect $f(\vec{x}) - y$ of $\mathcal{F}_{verify}$ is computed directly from x and y, which have already been delivered by $\mathcal{S}$ to $\mathcal{F}_{verify}$ for $k \in \mathcal{C}$.
   - For all $k \notin \mathcal{C}$, $\mathcal{S}$ simulates $\mathcal{F}_{verify}$ outputting $(id, 1)$ and writes $mlc_i[k] \leftarrow 0$ for all $i \notin \mathcal{C}$. It takes 0 as the side-effect of $\mathcal{F}_{verify}$.

• **Finally**, $\mathcal{F}_{vmpc}$ outputs to each party P_i the set of parties $\mathcal{B}$ for which $\vec{m}^{*\ell}_{ij} \neq \vec{m}^{\ell}_{ij}$ 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. 27. The simulator $\mathcal{S}_{vmpc}$ for verifiable computations

– *Base:* Initially, there are the inputs $\vec{x}_i$ and the randomness $\vec{r}_i$ in the internal state of $\mathcal{F}_{vmpc}$. So far, for $i \notin \mathcal{C}$, $\mathcal{A}$ has no information about $\vec{x}_i$, $\vec{r}_i$, and for $i \in \mathcal{C}$ it expects $\vec{x}_i = \vec{x}^*_i$, $\vec{r}_i = \vec{r}^*_i$, where $\vec{x}^*_i$ is chosen by $\mathcal{A}$ itself, and $\vec{r}^*_i$ 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 internal state of $\mathcal{F}_{vmpc}$ is consistent with the view of $\mathcal{A}$.

– *Step:* In the real world, for each $i \notin \mathcal{C}$, $\mathcal{A}$ chooses all the messages m^{ℓ}_{ji} for $j \in \mathcal{C}$ that will be delivered to P_i. By induction hypothesis, the rest of the messages m^{ℓ}_{ji} for $j \notin \mathcal{C}$ and the inputs/randomness $\vec{x}_i$,$\vec{r}_i$ of the 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 $\vec{m}^{\ell+1} = C^{\ell}_{ij}(\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell}_{ni})$. In the inner state of $\mathcal{F}_{vmpc}$, the value $\vec{m}^{\ell+1}$ is computed in exactly the same way.

2. *The sets $\mathcal{B}_i$ of blamed parties.* $\mathcal{F}_{vmpc}$ computes all the messages $\vec{m}^{\ell}_{ij}$ and constructs the set $\mathcal{M}$ of parties j for whom $\vec{m}^{\ell}_{ij} \neq \vec{m}^{*\ell}_{ij}$, where $\vec{m}^{*\ell}_{ij}$ 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 $\mathcal{B}_0 = \{k \mid (\mathsf{cheater}, k)$ has come from $\mathcal{F}_{verify}$ in the execution phase $\}$. The ideal functionality $\mathcal{F}_{vmpc}$ 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 $(\mathsf{cheater}, j)$ has come from $\mathcal{F}_{verify}$ at some moment. By definition of

$\mathcal{F}_{verify}$, no $(\mathsf{cheater}, j)$ 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 simulated the work of $\mathcal{F}_{verify}$ in such a way that it outputs $(id, 1)$. 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 $\vec{m}^{\ell}_{ij}$ 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 P_i had succeeded for every C^{ℓ}_{ij}. For all $i, j \in [n]$, $\ell \in [r']$, i should have come up with the commitments $\vec{x}_i$, $\vec{r}_i$, $\vec{m}^{\ell}_{ij}$ such that $\mathcal{F}_{verify}$ outputs $(id, 1)$ on input (verify, id) for each output wire identifier id. The committed $\vec{x}_i$ are chosen by $\mathcal{A}$ in the beginning of the execution phase, the randomness $\vec{r}_i$ is coming from the same distribution as the randomness generated by $\mathcal{F}_{vmpc}$, the incoming messages $\vec{m}^{\ell}_{ji}$ are those that are treated by $\mathcal{F}_{vmpc}$ as being sent to P_i by P_j, and the outgoing messages $\vec{m}^{\ell}_{ij}$ are the same that are computed by $\mathcal{F}_{vmpc}$ (the messages moving between two corrupted parties have been chosen by $\mathcal{A}$). Hence if verification has succeeded, $\vec{m}^{\ell}_{ij} = C^{\ell}_{ij}(\vec{x}_i, \vec{r}_i, \vec{m}^1_{1i}, \ldots, \vec{m}^{\ell-1}_{ni})$ for all $i, j \in [n]$, $\ell \in [r']$, so $i \notin \mathcal{M}'$. □

Lemma 13. *Let Π_{vmpc} use the implementation of Π_{verify} that is built on top of Π_{pre}, Π_{TR}, and Π_{share}. Let the initial protocol defined by the circuits C^{ℓ}_{ij} have the following parameters (for one prover):*
- *it has r rounds;*
- *its largest ring is $\mathbb{Z}_{2^m}$;*
- *the number of transmitted bits of the protocol is M_c;*
- *the number of input bits is M_x;*
- *the number of randomness bits is M_r;*
- *the number of bit related gates (bit decomposition, ring extension) is N_b;*
- *the number of multiplication gates is N_m;*
- *the number of input and output wires in the circuits (excluding the intermediate wires) is N_w.*

Let λ be the number of bits used for randomness seeds. The upper bounds for costs of the preprocessing, execution, and postprocessing phases of Π_{vmpc} are given in Table 14 for the optimistic case, where the adversary does not attempt to cheat.

Proof. Let $N_g := N_b + N_m$. We have taken the numbers of communicated bits from the previously proven lemmata for Π_{verify}. In the optimistic case, $\mathcal{F}_{TR}$ works in the cheap mode. We show how Table 14 is filled.

Preprocessing cost. The total cost $\mathsf{vcost}^{N_b,N_m,m}_{pre}$ of generating precomputed tuples is taken from Lemma 8. The total cost $\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r} \oplus \mathsf{fwd}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r} \oplus \mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r}$ of generating the randomness is taken from Table 12. All the randomness for one prover can be generated in parallel using the same transmissions and forwardings, so M_r moves into the subindex of tr and fwd. Taking the costs of different $\mathcal{F}_{TR}$ operations from Table. 7, the number of rounds of $\mathsf{vcost}^{N_b,N_m,m}_{pre}$ is $\max(1, 1+2) + \max(2,2,2) = 5$, regardless of the parameters N_b, N_m, m, and it is $1+1+1 = 3$ for the randomness generation. Since the preprocessed tuples and the randomness can be generated in parallel, we get the total number of $\max(5,3) = 5$ rounds in the cheap mode of $\mathcal{F}_{TR}$. The total number of called operations is counted by putting together $\mathsf{vcost}^{N_b,N_m,m}_{pre}$ and $\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r} \oplus \mathsf{fwd}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r} \oplus \mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot M_r}$. The number of $\mathcal{F}_{TR}$ operations is counted as follows.

- *Transmit:* Each of the n parties receives its shares of initial precomputed tuples as a single message. The other $3nt$ transmissions come from generating λ and M_r, where the randomness is treated as a single M_r-bit value.
- *Forward:* The randomness is treated as a single M_r-bit value. There are nt forwarding for its shares.
- *Broadcast:* All broadcasts come from openings. The shares of initially opened κ tuples are broadcast as a single message for all tuples. Both openings of the pairwise verification can be also treated as a single broadcast message for all tuples. Since all these broadcasts are done simultaneously by the prover, all these values can be sent in a single broadcast.
There are also t openings coming from the λ-bit public randomness used by Π_{pre}. Each party P_j broadcasts n shares of r_j that it has generated. Since the parties have to agree on public randomness before the cut-and-choose step, these broadcasts cannot be parallelized with the previous one.

Execution cost. Before the execution starts, each input has to be committed. The total cost of input commitment $\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot M_x}$ is taken from Table 12, where all the M_x bits of one prover are committed in parallel, so M_x moves into subindex.

Table 14. Costs of different phases of Π_{vmpc} for one prover in $\mathbb{Z}_{2^m}$

phase	# rounds (total)	$\mathcal{F}_{TR}$ op	#ops	#rounds	# bits
pre	5	transmit	$n + 2nt$	1	$n \cdot \mathsf{sh}_n \cdot m(\mu(N_b m + 3N_m) + \kappa(m+3))$ $+nt \cdot \mathsf{sh}_n \cdot \lambda$ $+nt \cdot \mathsf{sh}_n \cdot M_r$
			nt	1	$nt \cdot \mathsf{sh}_n \cdot M_r$
		forward	nt	1	$nt \cdot \mathsf{sh}_n \cdot M_r$
		broadcast	t	2	$n \cdot \mathsf{sh}_n \cdot \lambda$
			1	2	$n(\mu-1)m \cdot (N_b + \mathsf{sh}_n \cdot 2N_m)$ $+n \cdot \mathsf{sh}_n \cdot (\mu-1)m \cdot (N_b m + N_m)$ $+n \cdot \mathsf{sh}_n \cdot m \cdot \kappa(m+3)$
exec	$1+r$	transmit	$n(1+r)$	$1+r$	$\mathsf{sh}_n \cdot (n \cdot M_x + M_c)$
post	3	transmit	n^2	1	$\mathsf{sh}_n \cdot n \cdot M_c$
			n^2	1	$\mathsf{sh}_n \cdot n \cdot M_c$
		forward	n^2	1	$\mathsf{sh}_n \cdot n \cdot M_c$
		broadcast	1	2	$N_g \cdot 2m + n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$

The M_c bits of the original communication are transmitted in r rounds. On each round, up to $n-1$ distinct transmissions may take place for each party, since it may send something to $n-1$ other parties. Treating the final outputs as a part of these M_c bits, we also accept that a party may send values "to itself", so the upper bound is rn.

Postprocessing cost. The verification cost comes from the execution of $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_msg}, id)$ and (verify, id). It consists of the following blocks:

Message commitments: The cost $\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot M_c} \oplus \mathsf{fwd}^{\otimes n}_{\mathsf{sh}_n \cdot M_c} \oplus \mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot M_c}$ is taken from Table 12. All the messages can be committed in parallel, similarly to inputs and randomness. Although M_c bits need to be delivered only to n parties, different messages should be approved by different receivers. This results in n^2 transmissions (delivering each of the n shares to each of the n senders), and all these messages need to be forwarded. Finally, the receiver confirms the shares by transmitting them back to the share holders. This procedure takes 3 rounds, and it can actually be reduced to 2 rounds if the n shares are first delivered to the receiver, followed by the receiver forwarding them to share holders.

Hint broadcast: The total number of bits $N_g \cdot 2m$ is taken from Lemma 9. All these bits are broadcast as a single message.

Alleged zero broadcast: The total number of bits $n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$ is taken from Lemma 10. Here we assume that all the outputs of the circuits are exactly the communication messages output by the circuits, so we do not introduce M_y. All n shares of the alleged zero vector are broadcast in parallel by the prover, so it can be treated as a single broadcast.

Putting together the hint broadcast and the alleged zero broadcast, we get one broadcast involving $N_g \cdot 2m + n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$ bits. Since message commitment can be done in parallel with these broadcasts, the verification takes 3 rounds in the optimistic mode. □

C.4 Proof of the Main Theorem

We are now ready to prove Theorem 2. We take Π_{vmpc} that is built on top of Π_{verify} (which is in turn using Π_{share}, Π_{pre}, and Π_{TR}).

Correctness. For estimating the correctness error, we need to count the total number of messages sent using $\mathcal{F}_{TR}$, including all the transmitted, forwarded, and broadcast messages. By message, we mean a bitstring that is signed with one signature. For this, we look at the Table 14 and sum up the total number of different $\mathcal{F}_{TR}$ calls. The total number of transmitted and broadcast messages for one prover is

$$\begin{aligned} N &= n + 3nt + nt + t + 1 + rn + n + 2n^2 + n^2 + 1 \\ &= 3n^2 + 4nt + (r+2)n + t + 2 \\ &\le 7n^2 + (r+3)n \ . \end{aligned}$$

For n provers, the upper bound on N is $7n^2(n+r+3)$. By Lemma 3, the error ε of the underlying Π_{TR} is bounded by $7n^2(n+r+3) \cdot \delta$. If we want to achieve error $2^{-\eta-1}$, we need $\delta \le 2^{-\eta-1-\log(7n^2(n+r+3))}$.

The other source of error is Π_{pre}. In order to achieve error at most $2^{-\eta-1}$, by Lemma 7 it is sufficient to take $\mu = 1 + \frac{\eta+1}{\log N_g} \le \eta$, and

$$\begin{aligned} \kappa &= \max(\{(n^{1/\mu}+1)(\eta+1), n^{1/\mu}+\mu-1\}) \\ &\le \max(\{(2^{-\eta-1}+1)\eta, 2^{-\eta-1}+\eta+1\}) \approx \eta \ , \end{aligned}$$

which we will need when estimating the cost of preprocessing phase. This gives us an upper bound for the total error $2 \cdot 2^{-\eta-1} = 2^{-\eta}$.

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

Cost. We combine the numbers of Table 14 with the costs of particular $\mathcal{F}_{TR}$ operations of Table. 7. Since the variables N_b, N_m, 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. 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 $\kappa' := n\kappa$. Let λ' be the number of bits used in a signature. We will still need to multiply the number of used $\mathcal{F}_{TR}$ operations by n in the pre- and postprocessing phases.

Preprocessing cost. In order to achieve the reported correctness, we took $\mu \leq \eta$ and $\kappa \leq \eta + 1$ (so $\kappa' \leq n(\eta+1)$). We will use these numbers for finding an upper bound on communication complexity.

- *Transmit:* The total number of bits per party is

$$n \cdot \mathsf{sh}_n \cdot m(\mu(N_b m + 3N_m) + \kappa'(m+3)) + nt \cdot \mathsf{sh}_n \cdot \lambda + 2nt \cdot \mathsf{sh}_n \cdot M_r \ .$$

Since N_b, N_m are already counted for all n provers, and the same seed λ can be used with different randomness generators, this number is not multiplied by n. The total number of independent transmissions that need a signature is $(n + 3nt)$ for each prover. Cheap mode transmission adds the overhead of a λ'-bit signature and for each transmission *round* it adds n^2 bits related to "no complaints" notification exchanged between the parties.
Using the upper bounds for μ and κ' presented before, we get an upper bound of total bit communication for all n provers:

$$\begin{aligned}\mathsf{nb}^{\mathsf{tr}}_{pre} &= \mathsf{sh}_n \cdot n\eta m(mN_b + 3N_m) \\ &+\mathsf{sh}_n \cdot n^2\eta m(m+3) \\ &+\mathsf{sh}_n \cdot (nt \cdot \lambda + 2nt \cdot M_r) \\ &+n(n+3nt)\lambda' + 2n^2 \ .\end{aligned}$$

- *Forward:* The total number of forwarded bits is $nt \cdot \mathsf{sh}_n \cdot M_r$. From Table. 7, the cost of forwarding is twice the cost of a transmission. There are nt forwardings for each party, so the total number of bits is

$$\mathsf{nb}^{\mathsf{fwd}}_{pre} = 2\mathsf{sh}_n \cdot nt \cdot M_r + 2n \cdot nt \cdot \lambda' + n^2 \ .$$

- *Broadcast:* The total number of broadcast bits is $n(\mu-1)m \cdot (N_b + \mathsf{sh}_n \cdot 2N_m) + n \cdot \mathsf{sh}_n \cdot (\mu-1)m \cdot (N_b m + N_m) + n \cdot \mathsf{sh}_n \cdot m \cdot \kappa'(m+3)$ for the tuple verification, and $n \cdot \mathsf{sh}_n \cdot \lambda$ for agreement on public randomness. The realization of broadcast that we use multiplies this number of bits by n^2. Using the same inequalities as for transmission case, and moving N_b deeper into the brackets, we get

$$\begin{aligned}\mathsf{nb}^{\mathsf{bc}}_{pre} &= \mathsf{sh}_n \cdot n^3\eta m(mN_b + 3N_m) + n^3\eta m N_b \\ &+\mathsf{sh}_n \cdot n^4\eta m(m+3) + \mathsf{sh}_n \cdot n^3\lambda \\ &+n^2(t+1)\lambda' + 2n^2 \ .\end{aligned}$$

Summing together $\mathsf{nb}^{\mathsf{tr}}_{pre} + \mathsf{nb}^{\mathsf{fwd}}_{pre} + \mathsf{nb}^{\mathsf{bc}}_{pre}$, putting all the non-leading terms into o, treating λ, λ' as constants, and assuming for simplicity $n \leq \min(N_b, N_m)$ (each party computes at least one bit decomposition and one multiplication gate), and $\lambda \leq \eta$, we get the total number of bits upper bounded by

$$\mathsf{nb}_{pre} = \mathsf{sh}_n \cdot 4n^2(n\eta m(N_b m + 3N_m) + M_r) + o(n^3\eta m N_b) \ .$$

Execution cost. There are $rn + n$ transmissions per party. Since M_x and M_c are already estimated for all n parties, the cost of this phase is $\mathsf{sh}_n \cdot (n \cdot M_x + M_c) + (r+1)(n^2 \cdot \lambda' + n^2)$. Treating λ' as constant, we may write the total cost simply as

$$\mathsf{nb}_{exec} = \mathsf{sh}_n \cdot (n \cdot M_x + M_c) + o(rn^2) \ .$$

Postprocessing cost. Translating the values of Table 14 to communication gives us the following costs:

- *Transmit:* $2\mathsf{sh}_n \cdot n \cdot M_c + 2n^2\lambda' + 2n^2$.
- *Forward:* $2\mathsf{sh}_n \cdot n \cdot M_c + 2n^2\lambda' + n^2$.
- *Broadcast:* $n^2(N_g \cdot 2m + n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c) + \lambda') + n^2$.

Treating λ' as a constant, and assuming $n \leq N_g$ (each party computes at least one non-linear gate), we may write the cost as

$$\mathsf{nb}_{post} = \mathsf{sh}_n \cdot (2n^3 N_g m + n^2 M_c) + o(n^2 N_g m) \ .$$

Cheating overhead. In the pessimistic setting, if any party attempts to cheat, the preprocessing phase aborts. In the other phases, $\mathcal{F}_{TR}$ starts working in its expensive mode. As can be seen from Table 7, since the execution phase only involves transmissions, the number of rounds at most doubles, and the total communication increases up to $2n$ times. However, the overhead of the verification phase may be larger, due to the use of expensive broadcast.

We note that, if the cheap broadcast fails, and the expensive one should be called instead, then any party that had to abort is able to prove the fact that the sender was dishonest. Namely, it may use $\mathcal{F}_{transmit}$ to reveal the message that it received to all parties. In this way, we may assume that a covert adversary will not cheat with the broadcasts, since it will be caught if it does so. $\square$

Discussion. Treating the number of parties as a constant, we get the following complexities of different phases (in the optimistic setting):

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

If replicated secret sharing is used, then for $n = 5$, the constant of O is already quite large due to the exponential nature of share cost sh_n. Shamir's secret sharing can be used to solve the problem, but then we lose the advantage of computation in 2^m, thus being unable to verify bit decomposition gates directly (the construction still works for multiplication in $\mathbb{Z}_p$ for a prime p). The gates involving bit decomposition provide additional multiplicative overhead of m, where 2^m is the size of the ring in which the computation takes place. Otherwise, all the overheads are linear. We conclude that our mechanism is most suitable for $n = 3$ and computations over 2^m.