Preprocessing-Based Verification of Multiparty Protocols with Honest Majority

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

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

Abstract. This paper presents a generic method for turning passively secure protocols into protocols secure against covert attacks, adding an offline preprocessing and a cheap post-execution verification phase. The execution phase, after which the computed result is already available to the parties, has only negligible overhead.
Our method uses shared verification based on precomputed multiplication triples. Such triples are often used to make the protocol execution itself faster, but in this work we make use of these triples especially for verification. The verification preserves the privacy guarantees of the original protocol, and it can be straightforwardly applied to protocols over finite rings, even if the same protocol performs its computation over several distinct rings at once.

Introduction

Suppose that several distinct parties communicating over a network want to solve a common problem. As far as the parties trust each other, the task is trivial, but it becomes more difficult if it is not the case. It is known that such a computation can be performed in a manner that the participants only learn their own outputs and nothing else [1], 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 [2–5], as well as various protocols optimized to solve concrete problems [6–9].

Initially, two main kinds of adversaries are taken into account: passive and active. The highest performance and greatest variety is achieved for protocols secure against passive adversaries. However, in practice one would like to achieve a stronger security property. Since achieving security against active adversaries is expensive, some intermediate adversary classes (between passive and active) have been introduced.

In practice, it would often be sufficient that an active adversary would not be detected immediately during the computation, but probably later, after the computation has already been finished. Hence ideas of verifiable computation (VC) [10] are applicable to SMC. In general, VC is a method allows a weak

client to outsource a computation to a more powerful server that accompanies the computed result with a proof of correct computation that is relatively easy for a weak client to verify. Similar ideas can be used to strengthen protocols secure against passive adversaries. Namely, after executing the protocol, the parties can prove to each other that they have correctly followed the protocol, with little overhead.

This work is based on distributed verification, where the intermediate computation values of the prover are shared amongst a set of verifier parties where at least one verifier is honest. Although it is pretty trivial for the verifiers to repeat the same computation on these shares in a secret shared way, we do not want to execute the same protocol on the same inputs again, as this would be senseless. Instead, we make use of the fact that, after the initial protocol execution, the prover already knows all the intermediate values, and hence can make the verifiers' work significantly easier. In our construction, we apply the ideas of precomputation of multiplication triples [11] (*Beaver triples*), and combine them with linear secret sharing, which we use also for commitments. In general, using Beaver triples allows to make the protocol faster due to substituting multiplications with declassifications of random values, which is much cheaper. In our settings, the Beaver triples will be used in the verification process, and since the prover already knows which values will have to be declassified, it may publish all of them already *before* the verification starts. The correctness of these published values will be also verified in just one round.

We end up with a protocol transformation that makes the executions of any protocol (and not just SMC protocols) verifiable afterwards. Our transformation commits the randomness (this takes place offline), inputs, and the communication of the participants. The commitments are cheap, being based on digital signatures and not adding a significant overhead to the execution phase. The results of the protocol are available after the execution, and the verification can take place at any time after the execution.

The verification itself requires the prover to send around $O(|C|)$ information, but that can be done in one round. The remaining verification is local. The verification results of size $O(|C|)$ (in general, much less than $|C|$) are published by the verifiers in one round.

We present our protocol transformation as a functionality in the universal composability (UC) framework [12]. After reviewing related work in Sec. 1, we describe the ideal functionality in Sec. 2 and its implementation in Sec. 4. Before the latter, we give an overview of the existing building blocks we use in Sec. 3. We estimate the computational overhead of our transformation in Sec. 5.

1 Related Work

Our protocol transformation converts a protocol secure against a passive adversary to a protocol secure against *covert* adversary [13] that is prevented from deviating from the prescribed protocol by a non-negligible chance of getting caught. In our case, the probability of being caught is negligible, based on the

properties of underlying message transmission functionality (signatures), hash functions, and the protocols that generate offline preshared randomness.

Our transformation is very close to to [14], which is in turn similar to [15]. In [15], several instances of the initial protocol are executed, where only one instance is run on real inputs, and the other one on randomly generated shares. No party should be able to distinguish the protocol executed on real inputs from the protocol executed on random inputs. In the end, the committed traces of the random executions are revealed by each party, and everyone may check if a party acted honestly in the random executions. This way, in the beginning all the inputs must be reshared, and the computation must leak no information about the inputs, so that no party can guess which inputs are real and which are random.

In [14], the intermediate values of the circuit computed by each party are shared amongst the other parties using Shamir sharing. Afterwards, the correctness of these shares is verified by a linear probabilistically checkable proof (LPCP). This transformation can be applied to an arbitrary initial protocol, but computing the proof is expensive for the prover.

Similarly to [14], the goal of our transformation is to provide security against a certain form of active attackers. Currently, one of the best sets of SMC protocols secure against active adversaries is SPDZ [4,16]. Its performance is achieved through extensive offline precomputations, which involve Beaver triple generation. Similarly to several other protocol sets, SPDZ provides only a minimum amount of protocols to cooperatively evaluate an arithmetic circuit. A form of post-execution verifiability has been proposed for SPDZ [17]. Differently from SPDZ, we use Beaver triples not for the computation itself, but only for the final verification (which does not exclude the possibility of using triples also in the execution phase).

We have chosen [14] as the basis for our work. In this work, we use the same transmission functionality as in [14] (which has been initially motivated by an analogous functionality from [15]), but instead of LPCP we are going to use Beaver triples, which allow to apply the verification also to computation over rings.

2 Ideal Functionality

We specify our verifiable execution functionality in the universal composability (UC) framework [12]. We have n parties (indexed by $[n] = \{1, \ldots, n\}$), where $\mathcal{C} \subseteq [n]$ are corrupted for $|\mathcal{C}| = t < n/2$ (we denote $\mathcal{H} = [n] \backslash \mathcal{C}$). The protocol has r rounds, where the ℓ-th round computations of the party P_i, the results of which are sent to the party P_j, are given by an arithmetic circuit C_{ij}^{ℓ} over rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$. We define the following gate operations for such a circuit:

- The operations + (addition) and $*$ (multiplication) over rings $\mathbb{Z}_{n_1}, \ldots, \mathbb{Z}_{n_K}$ ($n_i < n_j$ for $i < j$).
- The operations trunc and zext are between rings. Let $x \in \mathbb{Z}_{n_x}$, $y \in \mathbb{Z}_{n_y}$, $n_x < n_y$.

  - $x = \mathsf{trunc}(y)$ computes $x = y \mod n_x$, going from a larger ring to a smaller ring.
  - $y = \mathsf{zext}(x)$ takes x and uses exactly the same value $y = x$ in $\mathbb{Z}_{n_y}$. If the ring sizes are powers of 2, it can be treated as taking the $\log n_x$ bits of x and extending them with zero bits to $\log n_y$ bits.
- The operation bits from an arbitrary ring $\mathbb{Z}_n$ to $(\mathbb{Z}_2)^{\log n}$ performs a bit decomposition. Although bit decomposition can be performed by other means, we introduce a separate operation, as it is reasonable to implement a faster verification for it.

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

Throughout this paper, we assume that the cardinalites of the rings are powers of two. Taking a ring of arbitrary size that is not a power of 2 is also possible, but that would be less efficient applying the methods proposed in this paper. All the results still hold without any efficiency loss if the computation is performed in a single finite field for which no bit operation support is needed.

In [14], the proof was based on the fact that Shamir sharing can be used with threshold. Consistency of Shamir sharing ensured that the set of all-honest verifiers that satisfy the threshold (which existed due to honest majority assumption) indeed accepts the proof. One significant disadvantage of rings of powers of 2 is that we cannot use Shamir secret sharing for verification, and additive secret sharing should be used instead. In this case, the prover has to repeat the proof with each subset of $t + 1$ verifiers separately (for efficiency gain, we may exclude the sets containing the prover itself), in order to ensure that at least one of the verifier sets consisted of honest provers only. The proof succeeds if and only if the outcomes of all the verifier sets are satisfiable. The number of verification sets is thus exponential in the number of parties, but it can still be reasonable for a small number of parties. Note that we require using the additive sharing scheme only in the verification phase, and it does not matter whether the original protocol that we are verifying has used any secret sharing at all.

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

We see that $\mathcal{M}$ is the set of parties deviating from the protocol. Our verifiability property is very strong as *all* of them will be reported to *all* honest parties. Even if only *some* rounds of the protocol are computed, all the parties that deviated from the protocol in completed rounds will be detected. Also, no

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

Fig. 1: The ideal functionality for verifiable computations

honest parties (in $\mathcal{H}$) can be falsely blamed. We also note that if $\mathcal{M} = \emptyset$, then $\mathcal{A}_S$ does not learn anything that a semi-honest adversary could not learn.

3 Building Blocks

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

Message transmission. For message transmission between parties, we use functionality $\mathcal{F}_{tr}$ [14, 15] which allows one to prove to third parties which messages one received during the protocol, and to further transfer such revealed messages. We use the definition of [14] that differs from Damgård et al.'s [15] $\mathcal{F}_{transmit}$ by supporting the forwarding of received messages, as well as broadcasting as a part of the outer protocol. The definition of the ideal functionality of $\mathcal{F}_{tr}$ is shown on Fig. 2. The real implementation of the transmission functionality is built on top of signatures. This makes the implementation very efficient, as hash trees allow several messages (sent in the same round) to be signed with almost the same computation effort as a single one [18], and signatures can be verified in batches [19]. An implementation of $\mathcal{F}_{tr}$ is given in [20].

Beaver triples. Beaver multiplication triples [11] are triples of values (a, b, c) in a ring $\mathbb{Z}_n$, such that $a, b \xleftarrow{\$} R$, and $c = a * b$. Precomputing such triples can be used to linearize multiplications. For example, if we want to multiply $x * y$, and a triple (r_x, r_y, r_{xy}) is already precomputed and preshared, we may first compute and publish $x' := x - r_x$ and $y' := y - r_y$ (x' and y' leak no information about x and y), and then compute the linear combination

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

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

$x * y = (x' + r_x)(y' + r_y) = x'y' + r_x y' + x' r_y + r_x r_y = x'y' + r_x y' + x' r_y + r_{xy}$. Differently from standard usage (like in SPDZ), we do not use these triples in the original protocol, but instead use them to simplify the verification phase. Since the preprocessing phase does not count, we may generate such triples using any actively secure protocol, for example similarly to SPDZ [4]. The ideal functionality is given in Fig. 3. In addition to Beaver triple generation, $\mathcal{F}_{pre}$ also generates and shares random ring elements and random bits, which can also be done in the preprocessing phase (differently from [14], where randomness sharing was not treated as preprocessing).

Implementing $\mathcal{F}_{pre}$ is actually cheap if we use honest majority assumption. Below are given some propositions (without formal proofs).

Beaver triple generation: In our verification, the prover P will know the precise values of all Beaver triple shares anyway. Let n be the number of required triples. P generates $2n + 3\eta$ triples for a security parameter η and shares them amongst all the other parties. After that, some randomly chosen η of those triples are revealed. If all of them are correct, then only negligible fraction of the remaining $2n + 2\eta$ triples is incorrect. However, we rather want not a negligible fraction of triples to be incorrect, but want to have all of them correct except with negligible probability. For this, the triples are randomly divided to n pairs, where in each pair one triple is sacrificed to check the other one, as it is done in SPDZ [4]. This uses up one triple from each pair, and any n of the $n+\eta$ remaining

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

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

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

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

Randomness distribution: On input (rnd, j, id) from M_i, check if $rnd[id]$ exists. If it does, take $(r^1, \ldots, r^n) := rnd[id]$. Otherwise, generate a ring element $r \xleftarrow{\$} \mathbb{Z}_{n(id)}$ and share it to r^k. Assign $rnd[id] := (r^1, \ldots, r^n)$. If $j \neq i$, send r^i to M_i. Otherwise, send $(r^1, \ldots, r^n)$ to M_i.

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

triples may now be used in the verification. The prover may cheat only if two wrong triples get into the same pair, but this probability is negligible since the fraction of such triples is small. Differently from SPDZ, we do not need to use any homomorphic encryption for triple generation, since we allow the prover to know them.

Randomness Using honest majority assumption, all the randomness can be generated similarly to [14]. The only question is how to ensure that the random bits are indeed bits if they are generated in a larger ring. Similarly to Beaver triple generation, the simplest way is generate $n + \eta$ bits and reveal randomly η of them. If we want the result to be 100% correct, performing checks of the form $b^2 - b = 0$ can be done, using a Beaver triple for computing each $b * b$ product.

Verifying Basic Operations.

If we need to compute $z := \mathsf{trunc}(x)$, we can *locally* convert the shares over the larger ring to shares over the smaller ring, which is correct as the sizes of the rings are powers of 2, and so the size of the smaller ring divides the size of the larger ring. However, if we need to compute $z := \mathsf{zext}(x)$, then we cannot just covert the shares of committed z locally, as zext is not an inverse of trunc, and we need to ensure that all the excessive bits of z are 0.

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

1. The bit decomposition operation $(z_0, \ldots, z_{n-1}) := \mathsf{bits}(z)$:
   check $z = z_0 + z_1 \cdot 2 + \cdots + z_{n-1} 2^{n-1}$;
   check $\forall j : z_j \in \{0, 1\}$.

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

Now all the ring-specific operations have been reduced to linear combinations and equality checks. The linear combinations can be computed locally on shares, and the equalities verified succinctly in parallel. However, the initial circuit may still contain some multiplication gates that cannot be computed locally and hence would provide a large computational overhead for the verifiers. We would like to get rid of all the multiplications, and we do it using Beaver triples.

Consider a circuit C_{ij}^{ℓ} being verified. For each multiplication gate, a Beaver triple is generated in the corresponding ring $\mathbb{Z}_{2^n}$. The triple is known by the prover, and it is used only in the verification, but not in the computation itself. The triple generation is performed using an ideal functionality $\mathcal{F}_{pre}$ (see Fig. 3) that generates Beaver triples and shares them amongst the parties. Additionally, this functionality generates and shares random bits, which will be used similarly to Beaver triples: at some moment, b' is published, such that $b = (b' + r_b) \mod 2$. These random bits are not used in multiplication, and they are used to ensure that b is a bit. Namely, if $b' = 0$, then $b = r_b$, and $b = 1 - r_b$ otherwise. If r_b is indeed a bit (which can be proved in the preprocessing phase), then b is also a bit.

Theorem 1. *Given a Beaver triple generation protocol with output error ε, any n-party r-round protocol Π can be transformed into an n-party $(r + 6)$-round protocol Ξ in the $\mathcal{F}_{tr}$-$\mathcal{F}_{pre}$-hybrid model, which computes the same functionality as Π and achieves covert security against adversaries statically corrupting at most $t < n/2$ parties, where the cheating of any party is detected with probability at least $(1-\varepsilon)$. If Π is δ-private against passive adversaries statically corrupting at most t parties, then Ξ is δ-private against cover adversaries. Under active attacks by at most t parties, the number of rounds of the protocol may at most double.*

Thm. 1 is proved by the construction of the real functionality Sec. 4, as well as the simulator presented in Appendix. A.

4 Real Functionality

The protocol Π_{vmpc} implementing $\mathcal{F}_{vmpc}$ consists of n machines $M_1, \ldots, M_n$ doing the work of parties $P_1, \ldots, P_n$, and the functionality $\mathcal{F}_{tr}$. The internal state of each M_i contains a bit-vector mlc_i of length n where M_i marks which other parties are acting maliciously. Some t of n parties are assigned to be *verifiers*, and the set of such parties is denoted by $\mathcal{V}$. The goal of the prover is to prove its

Circuits: M_i gets from $\mathcal{Z}$ the message $(\mathsf{circuits}, i, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ and sends it to $\mathcal{A}$.
Beaver triple generation Let id be the identifier of a multiplication gate of M_i, where both inputs are private. Each party M_k sends a query (beaver, i, id) to $\mathcal{F}_{pre}$. The prover M_i receives all the shares $(r_x^1, \ldots, r_x^{t'}, r_y^1, \ldots, r_y^{t'}, r_{xy}^1, \ldots, r_{xy}^{t'})$, and each verifier just the shares (r_x^k, r_y^k, r_{xy}^k).
Random bit generation Let id be the identifier of a circuit wire that needs a proof of correctness of its bit decomposition. For all $k \in \mathcal{V}$, M_k sends a query (bit, i, id) to $\mathcal{F}_{pre}$. The prover M_i receives all the shares $(\boldsymbol{b}^1, \ldots, \boldsymbol{b}^{t'})$, and each verifier just the share $\boldsymbol{b}^k$. Let $\bar{\boldsymbol{b}}_i^k$ be the vector of all such bit shares of the prover M_i issued to M_k.
Randomness generation and commitment: Let id be the identifier of a circuit wire that takes randomness as an input. For all $k \in \mathcal{V}$, M_k sends a query (rnd, i, id) to $\mathcal{F}_{pre}$. The prover M_i receives all the shares $r^1, \ldots, r^{t'})$, and each verifier just the share r^k. Let $\boldsymbol{r}_i^k$ be the vector of all such bit shares of the prover M_i issued to M_k.

Fig. 4: Preprocessing phase of the real functionality

Input commitments: M_i with $i \in \mathcal{H}$ [resp. $i \in \mathcal{C}$] gets from $\mathcal{Z}$ [resp. $\mathcal{A}$] the input $\boldsymbol{x}_i$ and shares it to n vectors $\boldsymbol{x}_i^1, \ldots, \boldsymbol{x}_i^{t'}$. For each $k \in \mathcal{V} \setminus \{i\}$, M_i sends to $\mathcal{F}_{tr}$ $(\mathsf{transmit}, (\mathsf{x\_share}, i, k), \boldsymbol{x}_i^k)$ for M_k.
At any time: if $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{tr}$, M_i writes $mlc_i[j] := 1$ and goes to the accusation phase.

Fig. 5: Initialization phase of the real functionality

honestness to each subset of t verifiers (the number of proofs is thus exponential in the number of parties). Due to the honest majority assumption, at least one of these subsets consists only of honest provers. The proof is accepted iff all the verifier subsets accept the proof. Hence our construction has to ensure that no set containing any dishonest verifiers may accuse an honest prover.

The protocol Π_{vmpc} runs in six phases: preprocessing, initialization, execution, message commitment, verification, and accusation. For simplicity, we describe each phase as it is defined for one proof (for one fixed set $\mathcal{V}$). These phases are exactly the same for all the subsets of $t' := t + 1$ verifiers, sharing no unique randomness, and hence they can be treated as independent proofs.

Preprocessing. This is a completely offline preprocessing phase that can be performed before any inputs are known. First of all, the circuits C_{ij}^{ℓ} are constructed and shown to the adversary. For the prover M_i, a Beaver triple is constructed and shared for each multiplication gate of C_{ij}^{ℓ}. In addition, N random bits are generated for each value that requires a bit decomposition in a ring of size 2^N. All these values are generated using $\mathcal{F}_{pre}$. The randomness vectors $\boldsymbol{r}_i$ used in the initial protocol are committed for each M_i also using $\mathcal{F}_{pre}$. This phase is given in Fig.4.

Initialization. In the initialization phase, the inputs $\boldsymbol{x}_i$. This phase is given on Fig.5. Note that the consistency of input shares is not verified. Intuitively, it can be only more difficult for the prover to prove the correctness of his computation for several distinct inputs at once. The simulator (in Appendix A) proves it more formally.

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

Fig. 6: Execution phase of the real functionality

Message sharing: As a sender, M_i shares $\boldsymbol{c}_{ij}^{\ell}$ to $\boldsymbol{c}_{ij}^{\ell k}$. For each $k \in \mathcal{V}$, M_i sends to $\mathcal{F}_{tr}$ the messages $(\mathsf{transmit}, (\mathsf{c\_share}, \ell, i, j, k), \boldsymbol{c}_{ij}^{\ell k})$ for M_j.
Public values: The prover M_i constructs the vector $\boldsymbol{b}_i^{\ell}$ which denotes which entries of communicated values of $\bar{\boldsymbol{b}}_i^{\ell}$ (related to communication values) should be flipped. Let $\boldsymbol{p}_i^{\ell}$ be the vector of the published values c' such that $c = (c' + r_c)$ is a masked communication value. M_i sends to $\mathcal{F}_{tr}$ a message $(\mathsf{broadcast}, (\mathsf{communication\_public}, \ell, i), (\boldsymbol{p}_i^{\ell}, \boldsymbol{b}_i^{\ell}))$.
Message commitment: upon receiving $((\mathsf{c\_share}, \ell, i, j, k), \boldsymbol{c}_{ij}^{\ell k})$ and $(\mathsf{broadcast}, (\mathsf{communication\_public}, \ell, i), (\boldsymbol{p}_i^{\ell}, \boldsymbol{b}_i^{\ell}))$ from $\mathcal{F}_{tr}$ for all $k \in \mathcal{V}$, the machine M_j checks if the shares correspond to $\boldsymbol{c}_{ij}^{\ell}$ it has already received. If only $c_{ij}^{\ell s\prime}$ is published for some $c_{ij}^{\ell s}$, then it checks $c_{ij}^{\ell s} = c_{ij}^{\ell s\prime} + r_c$ for the corresponding preshared randomness r_c (related to the Beaver triple). If something is wrong, M_j sends $(\mathsf{publish}, (\mathsf{message}, \ell, i, j))$ to $\mathcal{F}_{tr}$, so now everyone sees the values that it has actually received from M_i, and each (uncorrupted) M_k should now use $\boldsymbol{c}_{ij}^{\ell k} := \boldsymbol{c}_{ij}^{\ell}$. If the check succeeds, then M_i sends to $\mathcal{F}_{tr}$ $(\mathsf{forward}, (\mathsf{c\_share}, \ell, i, j, k))$ for M_k for all $k \in \mathcal{V} \setminus \{i\}$.

Fig. 7: Message commitment phase of the real functionality

Execution. The parties run the original protocol as before, just using $\mathcal{F}_{tr}$ to exchange the messages. This phase is given on Fig.6. If at any time in some round ℓ the message $(\mathsf{corrupt}, j)$ comes from $\mathcal{F}_{tr}$ (all uncorrupted machines receive it at the same time), the execution is cut short, no outputs are produced and the protocol continues with the commitment phase.

Message commitment. In the message commitment phase, all the n parties finally commit their sent messages $\boldsymbol{c}_{ij}^{\ell}$ for each round $\ell \in [r']$ by sharing them to $\boldsymbol{c}_{ij}^{\ell k}$ and sending these shares to the other parties. This phase is given on Fig. 7. Let $\boldsymbol{v}_{ij}^{\ell} = (\boldsymbol{x}_i \| \boldsymbol{r}_i \| \boldsymbol{c}_{1i}^{1} \| \cdots \| \boldsymbol{c}_{ni}^{\ell-1} \| \boldsymbol{c}_{ij}^{\ell})$ be the vector of inputs and outputs to the circuit C_{ij}^{ℓ} that M_i uses to compute the ℓ-th message to M_j. After this phase, M_i has shared $\boldsymbol{v}_{ij}^{\ell}$ among the t' verifier parties. Let $\boldsymbol{v}_{ij}^{\ell k}$ be the share of $\boldsymbol{v}_{ij}^{\ell}$ given to machine M_k.

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

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

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

Fig. 8: Verification phase of the real functionality

Verification phase.

The proving party publishes all the remaining public Beaver triple values, and all the remaining bits b'_{ij} for each bit b_{ij} that require a proof of being a bit (see Fig. 8). For each operation $z := \mathsf{zext}(x)$, where $z \in \mathbb{Z}_{2^{n_e}}$, the prover commits by sharing the value of z in the ring $\mathbb{Z}_{2^{n_e}}$.

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

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

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

The correctness of operations $(z_1, \ldots, z_{n_e}) := \mathsf{bits}(z)$, $z = \mathsf{zext}(x)$, and $z = \mathsf{trunc}(x)$ is verified as shown in Sec. 3. The condition $\forall j : \ z_j \in \{0, 1\}$ can be ensured as follows: using the bit r_{z_j} shared in the preprocessing phase, and z'_j published in the commitment phase, each party locally computes the share of $z_j \in \{0, 1\}$ as r_{z_j} if $z'_j = 0$, and $1 - r_{z_j}$ if $z'_j = 1$. In the case of zext, the verifiers compute the shares of y locally, and take the shares of z that are committed by the prover in the commitment phase. Now, the checks of the form $x - y = 0$ and $z_0 + z_1 \cdot 2 + \ldots + z_{n_e-1} \cdot 2^{n_e-1} - z = 0$ are left. Such checks are just linear combinations of the shared values. As the parties cannot verify locally if the shared value is 0, they postpone these checks to the last round.

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

Fig. 9: Accusation phase of the real functionality

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

Finally, the verifiers come up with the shares $\bar{c}_{ij}^{\ell k}$ of the values $\bar{c}_{ij}^{\ell}$ that should be the outputs of the circuits. The verifiers have to check $c_{ij}^{\ell} = \bar{c}_{ij}^{\ell}$, but they cannot do it locally since the shares $c_{ij}^{\ell k}$ and $\bar{c}_{ij}^{\ell k}$ can be different. Again, an online linear combination check is needed for each $c_{ij}^{\ell k}$.

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

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

Accusation. Finally, each party outputs the set of parties that it considers malicious. This short phase is given in Fig. 9.

The formal UC proof for the real functionality can be found in Appendix A.

5 Efficiency

In this section we estimate the overheads caused by our protocol transformation.

Let C be the circuit of the prover. Let n be the total number of parties, $t < n/2$ the number of corrupt parties, r the number of rounds, N_g the number of gates, N_w the number of wires, N_x the number of inputs (elements of a ring $\mathbb{Z}_N$), N_c the number of communicated ring elements, N_r the number of random ring elements, and $N_i = N_w - N_x - N_r - N_c$ the number of intermediate wires in the circuit. Then $|\boldsymbol{v}| = N_x + N_r + N_c$ (where $\boldsymbol{v}$ is the shared commitment, as shown in Sec. 4).

In the case of additive sharing, up to $\binom{n}{t+1}$ proofs have to be performed in parallel for each party. For simplicity, below we present the complexity of one such proof. The local computations all fit into $O(|C|)$, and so we do not count them. Let the set of $t + 1$ verifiers be fixed. Compared to the original protocol, for each M_i the proposed solution has the following network communication

overheads (measured in the number of ring elements, which may be a bit unfair since different elements may belong to different rings).

Preprocessing: Is done offline and does not depend on the inputs. Hence we do not take its complexity into account.

Initialization: Share one vector of length N_x (the input $\boldsymbol{x}$) amongst the $t+1$ verifiers in $N_x \cdot (t+1)$ network communication.

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

Message commitment: Send to each verifier party $(t+1) \cdot N_c$ ring elements, which represent the total communication commitment.

Verification: The prover now publishes the vectors $\boldsymbol{p}_i^\ell$ and $\boldsymbol{b}_i$ for each round ℓ. In the worst case, such a vector should be published for each wire, and hence the prover has to broadcast at most $2N_w$ ring elements. The simplest broadcast based on honest majority requires that all the communication should be distributed by each party to each other party two times, and hence there will be $n \cdot 2N_w + 2n^2 \cdot 2N_w$ communicated ring elements. All the verifiers then compute the circuit C on their shares locally and then compute and publish the final shares of $A\boldsymbol{x}$, which is up to N_w ring elements. In the case $t = 3$, they only publish one ring element, which is either $h(A\boldsymbol{x})$ or $h(-A\boldsymbol{x})$.

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

6 Conclusions and Further Work

We have proposed a scheme transforming passively secure protocols with honest majority to covertly secure ones. The protocol transformation is suitable to be implemented on top of some existing, highly efficient, passively secure SMC frameworks, especially those that use 3 parties and computation over rings of size 2^N. The framework will retain its efficiency, as the time from starting a computation to obtaining the result at the end of the execution phase will not increase. Also, the overheads of verification, which are proportional to the number of parties, will be rather small due to the small number of *computing parties* in all typical SMC deployments (the number of *input* and *result parties* may be large, but they can be handled separately).

References

1. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
2. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
3. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

4. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
5. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
6. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
7. Octavian Catrina and Sebastiaan de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2010.
8. Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.
9. Matthew K. Franklin, Mark Gondree, and Payman Mohassel. Communication-efficient private protocols for longest common subsequence. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *Lecture Notes in Computer Science*, pages 265–278. Springer, 2009.
10. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.
11. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Feigenbaum [21], pages 420–432.
12. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
13. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.
14. Peeter Laud and Alisa Pankova. Verifiable Computation in Multiparty Protocols with Honest Majority. In Sherman S. M. Chow, Joseph K. Liu, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings*, volume 8782 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2014.
15. Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In Daniele Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.
16. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
17. Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2014.

18. Ralph C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.
19. Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch verification of short signatures. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.
20. Peeter Laud and Alisa Pankova. Verifiable Computation in Multiparty Protocols with Honest Majority. Cryptology ePrint Archive, Report 2014/060, 2014. `http://eprint.iacr.org/`.
21. Joan Feigenbaum, editor. *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*. Springer, 1992.

A Simulator

In this section we prove that our protocol is as secure as $\mathcal{F}_{vmpc}$. We have to show that there exists a simulator that can translate the between the messages $\mathcal{F}_{vmpc}$ exchanges with the ideal adversary, and the messages the protocol in Fig. 5–9 exchanges with the real adversary over the network. We present the work of the simulator S in phases, coinciding with the phases of real functionality that it simulates to the adversary $\mathcal{A}$.

Preprocessing

Circuits: In the beginning, $\mathcal{F}_{vmpc}$ gets the messages $(\mathsf{circuits}, i, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ from $\mathcal{Z}$ for each party P_i and forwards them all to S. S delivers them to $\mathcal{A}$.

Beaver triple generation:. In the real functionality, the triples for x and y (used in the product $x*y$) are generated using $\mathcal{F}_{pre}$. From the name of dishonest parties, $\mathcal{A}$ sends the queries to S which has to simulate the behaviour of $\mathcal{F}_{pre}$. For the values known by dishonest parties, S generates the randomness in exactly the same way as $\mathcal{F}_{pre}$ does. The randomness r_x of honest parties cannot be randomly generated since S will have to publish the public masked value $x' = (x - r_x)$ at some point, but x is unknown to S. Since the malicious parties receive only t shares of r_x out of $t+1$, S just generates random uniform shares r_x^k for the malicious parties without thinking of what the remaining shares could be.

Randomness generation and commitment: The bit randomness, and also the randomness $\boldsymbol{r}_i$ used in the initial protocol execution, are both generated similarly to the Beaver triples. For each $i \in [n]$, $\mathcal{F}_{vmpc}$ sends $(\mathsf{randomness}, i, \boldsymbol{r}_i)$ to S.

Initialization.

Input commitments: For $i \in \mathcal{H}$, S should generate the shares of $\boldsymbol{x}_i$ by itself. At most t shares are sent to $\mathcal{A}$, and they look random due to additive sharing, so S generates them uniformly at random. Now S has to send $(\mathsf{input}, \boldsymbol{x}_i)$ to $\mathcal{F}_{vmpc}$ for each malicious party P_i. In the real functionality, the vectors $\boldsymbol{x}_i$ of malicious parties are committed as shares whose particular values are chosen by $\mathcal{A}$.

At any time: During all transmissions of the real functionality, S simulates the functionality $\mathcal{F}_{tr}$. If $(\mathsf{corrupt}, i)$ is output from $\mathcal{F}_{tr}$, S sends $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$ for $\mathcal{B}_0 = \{i \mid (\mathsf{corrupt}, i) \text{ has been output}\}$. The simulator S will follow it

up with messages $(\mathsf{blame}, i, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$ for all $i \in \mathcal{H}$. This position corresponds to the direct jump to the accusation phase in the real functionality.

Execution.

For each round ℓ: for each $i \in \mathcal{H}$ and $j \in [n]$, $\mathcal{F}_{vmpc}$ uses C_{ij}^{ℓ} to compute the message $\boldsymbol{m}_{ij}^{\ell}$. For all $j \in \mathcal{C}$, it sends $\boldsymbol{m}_{ij}^{\ell}$ to S. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it waits for $\boldsymbol{m}_{ji}^{\ell}$. S models the communication of M_j and M_i through $\mathcal{F}_{tr}$. It takes $\boldsymbol{m}_{ji}^{\ell} = \boldsymbol{c}_{ji}^{\ell}$.

After r rounds: S goes to the message commitment phase with $r' = r$.

At any time: If $(\mathsf{corrupt}, j)$ is output from $\mathcal{F}_{tr}$, S sends $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$ for $\mathcal{B}_0 = \{i \mid (\mathsf{corrupt}, i)$ has been output$\}$. In the real functionality, S goes to the message commitment phase with $r' = \ell - 1$.

Message commitment.

Message sharing: After the simulation of the initial protocol has finished, $\mathcal{F}_{vmpc}$ waits for the messages $\boldsymbol{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$. For each corrupt M_i, $\mathcal{A}$ generates the shares by itself. For each uncorrupted M_i, S has to give to $\mathcal{A}$ only t of the $t+1$ shares of each communication, so the knowledge of $\boldsymbol{c}_{ij}^{\ell}$ is unnecessary.

Public values: $\mathcal{A}$ comes up with the public values for malicious parties. S has to think out the values for the honest parties.

- For each value x used as an input of a multiplication gate, S knows the randomness shares r_x^k that were issued to the malicious parties. S now has to compute $x' = (x - r_x)$. Neither $\mathcal{A}$ nor S know the particular values of r_x^k distributed to the honest parties, but the sum of these shares r_x is definitely uniform at random since this is ensured by the preprocessing phase. Hence S just generates a random value x' and publishes it.
- The same is done to the bits. The bit randomness is shared in such a way that at most t shares are known to $\mathcal{A}$ and S, but there is still at least one share left. S just generates and reveals a random bit.

Message commitment: If $i \in \mathcal{C}$, and $\mathcal{A}$ provides inconsistent shares for some uncorrupted receiver M_j (including the public values $c_{ij}^{\ell\,\prime}$ for masked communication c_{ij}^{ℓ}), then S publishes $\boldsymbol{c}_{ij}^{\ell}$ through $\mathcal{F}_{tr}$, as a real honest receiver would do, and from the side of all honest parties it now assumes that $\boldsymbol{c}_{ij}^{\ell}$ is committed. If both $i, j \in \mathcal{C}$, then S defines $\boldsymbol{m}_{ij}^{\ell} = \sum_{k \in \mathcal{V}} \boldsymbol{c}_{ij}^{\ell k}$, so that $\boldsymbol{m}_{ij}^{\ell}$ corresponds to the view of the verifier parties $\mathcal{V}$. If $\mathcal{A}$ decides that M_j should complain about M_i and publishes $\boldsymbol{c}_{ij}^{\ell}$, then S takes $\boldsymbol{m}_{ij}^{\ell} = \boldsymbol{c}_{ij}^{\ell}$. S sends $\boldsymbol{m}_{ij}^{\ell}$ to $\mathcal{F}_{vmpc}$.

Remaining commitment: $\mathcal{A}$ chooses the remaining commitments for the malicious parties, and S generates the shares of the honest parties that should be given to the malicious party. S generates these shares uniformly at random, without knowing what the actual value of the remaining commitment is.

Verification.

Local computation: Each verifying party should compute the circuits of the proving party M_i on its local shares. $\mathcal{A}$ decides on the values $\boldsymbol{d}_{ij}^{\ell k}$ for dishonest parties. S holds all the shares that the malicious parties have sent. If $t > 3$, then $\boldsymbol{d}_{ij}^{\ell k}$ are published directly. if $t = 3$, then $h(\boldsymbol{d}_{ij}^{\ell k})$ is published instead.

– For the honest prover M_i, S knows that it is honest, and hence knows that it should be $\sum_{k\in V} \boldsymbol{d}_{ij}^{\ell k} = \mathbf{0}$. From the shares issued to malicious verifier, it constructs the shares $\boldsymbol{d}_{ij}^{\ell k}$ of the malicious verifiers (as if they acted honestly) and then computes $\boldsymbol{d} := \sum_{k\in\mathcal{V}\cap\mathcal{C}} \boldsymbol{d}_{ij}^{\ell k}$ as the sum of shares of the honest verifiers, sharing $\boldsymbol{d}$ randomly amongst $\mathcal{V}\cap\mathcal{C}$. If $t = 3$, then a single share $h_{ij}^{\ell k}$ is published by the only dishonest party, and S just takes $h_{ij}^{\ell k'} := h_{ij}^{\ell k}$ for the remaining honest verifier $k' \in \mathcal{V}\cap\mathcal{C}$.
– If the prover is malicious, then S uses the shares that were issued by $\mathcal{A}$ to the honest verifiers.

Complaints and final verification: If $\mathcal{A}$ decides that M_k refuses to broadcast, or the broadcast value is not equal to $(A_1\boldsymbol{x}_1^k\| \ldots \|A_K\boldsymbol{x}_K^k)$, then S broadcasts the complaint. Now $\mathcal{A}$ has the right to reveal the shares that have been sent to M_k. According to $\mathcal{F}_{tr}$ properties, the only case when it can falsify the shares is when M_i is corrupted. If it is not the case, then $\mathcal{A}$ may output only the shares that have actually been transmitted or forwarded by M_i, which are consistent with the valid commitment. For $t = 3$, S may analogously complain that the broadcast is not equal to $h(\pm(A_1\boldsymbol{x}_1^k\| \ldots \|A_K\boldsymbol{x}_K^k))$ (S may compute h itself since h is deterministic). Hence an honest M_i will not be blamed in any case. If both M_i and M_k are corrupted, then $\mathcal{A}$ may still provide a false proof for M_i. That is the reason why we repeat the proof with all sets of $t + 1$ parties and ensure that at least in one proof $\mathcal{V} \subseteq \mathcal{H}$.

If the proof of M_i has not failed yet due to misuse of $\mathcal{F}_{tr}$ (which is true at least for uncorrupted M_i), all the shares $(A_1\boldsymbol{x}_k^1\| \ldots \|A_K\boldsymbol{x}_K^k)$ for all $k \in V$ are published (in the case $t = 3$, $h((A_1\boldsymbol{x}_1^k\| \ldots \|A_K\boldsymbol{x}_K^k))$ and $h((A_1\boldsymbol{x}_1^{k'}\| \ldots \|A_K\boldsymbol{x}_K^{k'}))$ are published). S makes each uncorrupted verifier M_h act exactly like in the real protocol and checks if the published values are equal. If the check does not pass, S writes $mlc[h, i] := 1$ for each honest party P_h.

Accusation. $\mathcal{F}_{vmpc}$ computes all the messages $\boldsymbol{m}_{ij}^{\ell}$ and constructs $\mathcal{M}$. It is waiting for $(\mathsf{blame}, i, \mathcal{B}_i)$ from the adversary, such that $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. Let $\mathcal{B}_i' = \{j \mid mlc[i, j] = 1\}$. S defines $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}_i'$. First, we prove that $\mathcal{B}_i \subseteq \mathcal{C}$.

1. For each $j \in \mathcal{B}_0$, a message $(\mathsf{corrupt}, j)$ has come from $\mathcal{F}_{tr}$ at some moment. Due to properties of $\mathcal{F}_{tr}$, no $(\mathsf{corrupt}, j)$ can be sent for $j \in \mathcal{H}$. Hence $j \in \mathcal{C}$.
2. For each $j \in \mathcal{B}_i'$, the proof of M_j has not passed the final verification. For each honest M_j, S has chosen $\boldsymbol{d} := \sum_{k\in\mathcal{V}\cap\mathcal{C}} \boldsymbol{d}_{ij}^{\ell k}$ ($h_{k'} := h_k$ for $t = 3$), and the check passes since the sum of shares of all the verifiers is $\mathbf{0}$. Hence for an uncorrupted M_j the proof would always succeed, so $j \in \mathcal{C}$.

Secondly, we prove that $\mathcal{M} \subseteq \mathcal{B}_i$.

1. The first component of $\mathcal{M}$ is $\mathcal{B}_0$ for which the message $(\mathsf{stop}, \mathcal{B}_0)$ has been sent to $\mathcal{F}_{vmpc}$. S sends to $\mathcal{F}_{vmpc}$ the same $\mathcal{B}_0$ that is a subset of $\mathcal{B}_i$.
2. The second component $\mathcal{M}'$ of $\mathcal{M}$ are the machines M_i for whom inconsistency of $\boldsymbol{m}_{ij}^{\ell}$ happens in $\mathcal{F}_{vmpc}$. We show that if $M_i \notin \mathcal{B}_i$, then $M_i \notin \mathcal{M}'$. Suppose by contrary that there is some $M_i \in \mathcal{M}$, $M_i \notin \mathcal{B}_i$. No honest party

may get into $\mathcal{M}$, hence $M_i \in \mathcal{C}$. If $M_i \notin \mathcal{B}_i$, then the proof of M_i had succeeded for every C_{ij}^{ℓ}. For all $i, j \in [n]$, $\ell \in [r']$, M_i should have come up with $\boldsymbol{d}_{ij}^{\ell k}$ such that $\sum_{k \in \mathcal{V}} \boldsymbol{d}_{ij}^{\ell k} = \mathbf{0}$ ($h_{ij}^{\ell 1}$ and $h_{ij}^{\ell 2}$ such that $h_{ij}^{\ell 1} = h_{ij}^{\ell 2}$).
Consider now the proof in which $\mathcal{V} \subseteq \mathcal{H}$. There is at least one such set due to honest majority assumption. Since all the verifiers in this set are honest, they indeed use the shares that have been obtained during the protocol run. All these shares are known by S since all of them have passed through the malicious prover M_i. For $t > 3$, this immediately implies $(A_1\boldsymbol{x}_1 \| \ldots \| A_K\boldsymbol{x}_K) = \sum_{k \in \mathcal{V}} \boldsymbol{d}_{ij}^{\ell k} = \mathbf{0}$ For $t = 3$, if $h_{ij}^{\ell 1} = h_{ij}^{\ell 2}$, then, due to the collision-resistance of h, $\forall i: \ A_i\boldsymbol{x}_i^1 = -A_i\boldsymbol{x}_i^2 \iff A_i\boldsymbol{x}_i = \mathbf{0}$ with a probability p that depends on the properties of h.
The verifiers have simulated the computation on their shares locally on the subcircuits separated by $zext$ operations, bit decompositions, and Beaver triple masks. If $\forall A_i\boldsymbol{x}_i = 0$, this means that all these transition points have been computed correctly with respect to the committed inputs, randomness, and the communication. Hence $\boldsymbol{m}_{ij}^{\ell} = C_{ij}^{\ell}(\boldsymbol{x}_i, \boldsymbol{r}_i, \boldsymbol{m}_{1i}^1, \ldots, \boldsymbol{m}_{ni}^{\ell-1})$ for all $i, j \in [n]$, $\ell \in [r']$ and $M_i \notin \mathcal{M}'$.