Publicly Auditable Secure Multi-Party Computation

Carsten Baum, Ivan Damgård, and Claudio Orlandi

Department of Computer Science, Aarhus University,
{cbaum,ivan,orlandi}@cs.au.dk

Abstract. In the last few years the efficiency of secure multi-party computation (MPC) increased in several orders of magnitudes. However, this alone might not be enough if we want MPC protocols to be used in practice. A crucial property that is needed in many applications is that everyone can check that a given (secure) computation was performed correctly – even in the extreme case where all the parties involved in the computation are corrupted, and even if the party who wants to verify the result was not involved. An obvious example of this is electronic voting, but also in many types of auctions one may want independent verification of the result. Traditionally, this is achieved by using non-interactive zero-knowledge proofs.

A recent trend in MPC protocols is to have a more expensive preprocessing phase followed by a very efficient online phase, e.g., the recent so-called SPDZ protocol by Damgård et al. Applications such as voting and some auctions are perfect applications for these protocols, as the parties usually know well in advance when the computation will take place, and using those protocols allows us to use only cheap information theoretic primitives in the actual computation. Unfortunately no protocol of the SPDZ type supports an audit phase.

In this paper we formalize the concept of *publicly auditable secure computation* and provide an enhanced version of the SPDZ protocol where, even if all the servers are corrupted, anyone with access to the transcript of the protocol can check that the output is indeed correct. Most importantly, we do so without compromising the performance of SPDZ i.e., the cost of our online phase is the same as that of SPDZ, up to a small constant factor of about two.

Keywords: Efficient Multiparty Computation, Electronic voting, Public Verifiability.

Table of Contents

1 Introduction

The idea of computing on encrypted data is essentially as old as public-key cryptography [22] and in some sense precedes the introduction of secure two- and multi-party computation by Yao [23] and Goldreich, Micali and Widgerson [15].

During the last few years MPC has evolved from an academic topic to a practical tool. Several recent protocols (e.g. BeDOZa [4], TinyOT [20] and the celebrated SPDZ [10], [8]) achieve incredible performance for the actual function evaluation, even if all but one player is actively corrupted. This is done by pushing all the expensive cryptographic work into an offline phase and using only simple arithmetic operations during the online phase (note also that the offline phase is independent from the inputs and the circuit to be computed – only an upper bound on the number of multiplication gates is needed). Since these protocols allow to evaluate an arbitrary circuit (where a circuit represents a function over a field), one can in particular use these protocols to implement, for instance, a "shuffle-and-decrypt" operation for a voting application or the function that computes the winning bid in an auction. It is often the case that we know well in advance the time at which a computation is to take place, and in any such case, these protocols offer very good performances. In fact the computational work per player in the SPDZ protocol is comparable to the work one needs to compute the desired function in the clear, with no security.

However efficiency is not always enough: if the result we compute securely has large economic or political consequences, such as in voting or auction applications, it may be required that correctness of the result can be verified later. Ideally, we would want that this can done even if all parties involved in the computation are corrupted, and even if the party who wants to verify the result was not involved in the computation.

The traditional solution to this is to ask every player to commit to all his secret data and to prove in zero-knowledge for every message he sends, that this message was indeed computed according to the protocol. If a common reference string is available, we can use non-interactive zero-knowledge proofs, which allows anyone to verify the proofs and hence the result at any later time. However, this adds a very significant computational overhead, and would lead to a horribly inefficient protocol, compared to the online phase of SPDZ, for instance. On the other hand, as mentioned, none of the efficient protocols in the preprocessing model are auditable.

It is therefore natural to ask whether it is possible to achieve the best of both worlds and have *highly efficient MPC protocols with a non-cryptographic online phase that are auditable*, in the sense that everyone who has access to the transcripts of the protocol can check if the result is correct *even when all the servers are corrupted.* In this paper we answer this question in the affirmative.

1.1 Contributions and Technical Overview

The Model. In this work we will focus on client-server MPC protocols, where a set of parties (called the input parties) provide inputs to the actual working parties, who run the MPC protocol among themselves and make the output public. Note that the sets need not be distinct, and using standard transformations we can make sure that the servers do not learn the inputs nor the output of the computation (think of the inputs/output being encrypted or secret shared). We will focus on the setting of MPC protocols for dishonest majority (and static corruptions): as long as there is one honest party we can guarantee privacy of the inputs and correctness of the results, but we cannot guarantee termination nor fairness. We will enhance the standard network model with a "bulletin board" functionality. Parties are allowed to exchange messages privately, but our protocol will instruct them also to make part of their conversation public.

Auditable MPC. Our first contribution is to provide a formal definition of the notion of *publicly auditable MPC* as an extension of the classic formalization of secure function evaluation where we ask that the protocol implements an ideal functionality that receives inputs from the clients and computes the desired function. We require the usual security guarantees when there is at least one honest party, and in addition we ask that anyone, having only access to the transcript of the computation published on the bulletin board, can check the correctness of the output. This is formalized by introducing an extra, non-corruptible party (the

auditor) who can ask the functionality if the output was correct or not (of course, this only holds in the case where the computation did not abort). We stress that the auditor does not need to be involved (or even exist!) before and during the protocol. The role of the auditor is simply to check, once the computation is over, whether the output was computed correctly or not.
In Appendix A we discuss how this notion can be achieved using generic primitives[1].

SPDZ Recap. Given the motivation of this work, we are only interested in the notion of auditable MPC if it can be achieved efficiently. Therefore our starting point is one of the most efficient MPC protocols for arithmetic circuits with a cheap, information theoretic online phase, namely SPDZ.
In a nutshell SPDZ works as follows: At the end of the offline phase all parties hold additive shares of multiplicative triples (x, y, z) with $z = x \cdot y$. Now the players can use these preprocessed triples to perform multiplications using only linear operations over a the finite field (plus some interaction). Moreover, these linear operations can now be performed locally and are therefore essentially for free. However an adversary could send the honest parties a share that is different from what he received at the end of the offline phase. To make sure this is not the case, SPDZ adds information theoretic MACs of the form $\gamma = \alpha \cdot x$ to each shared value x, where both the MAC γ and the key α are shared among the parties. These MACs are trivially linear and can therefore "follow" the computation. Once the output is reconstructed, the MAC keys are also revealed and the MACs checked for correctness, and in the case the check goes through, the honest parties accept the output.

Auditable SPDZ. In order to make SPDZ auditable, we enhance each shared value x with a Pedersen commitment $g^x h^r$ to x with randomness r. The commitment key (g, h) comes from a common reference string (CRS), such that even if all parties are corrupted, those commitments are still binding. To allow the parties to open their commitments, we provide them also with a sharing of the randomness r (the parties already know a share of x). It is easy to see that this new "representation" of values is still linear and therefore does not disturb any of the SPDZ functionalities. It seems now that we did not achieve our goal, because the parties need to compute on the committed values and this requires exponentiations and other expensive operations. However, this is not the case! Computing parties will simply ignore their commitments (e.g., they only have to post them on the bulletin board) and it will be the job of the auditor to use the linear properties of the commitments to verify that each step of the computation was carried out correctly, using the commitments to the inputs, the multiplication triples, and the messages posted on the bulletin board. The only extra cost for the players is that now they have to compute every operation in parallel on the value x and on r as well – it is an interesting open problem to see if one could get rid of even this minor slowdown.
Clearly the "offline phase" of SPDZ needs to be modified as well, in order to produce the commitments to be used by the auditors and to satisfy the auditability requirement as well. In Chapter 5 we provide a description of the offline phase, and we will prove its security in Chapter 6.

1.2 Related Work

In publicly verifiable delegation of computation (see e.g. [14,13] and references therein) a computationally limited device delegates a computation to the cloud and wants to check that the result is correct. Verifiable delegation is useless unless verification is more efficient than the evaluation. Note that in some sense our requirement is the opposite: We want our workers to work as little as possible, while we are fine with asking the auditor to perform more expensive computation. External parties have been used before in cryptography to achieve otherwise impossible goals like fairness [17], but note that in our case *anyone can be the auditor* and the auditor does not need to be online while the protocol is executed. This is a qualitative difference with most of the other semi-trusted parties that appear in literature. A recent work [2] investigated an enhanced

[1] In a nutshell we can achieve auditable MPC, starting from a strong semi-honest protocol and then compiling it using NIZKoKs – a semi-honest MPC protocol alone would not suffice as we cannot force the parties to sample uniform randomness, nor can we trust them to force each other to do so by coin flipping when everyone is corrupted.

notion of covert security, that allows anyone to determine if a party cheated or not given the transcript of the protocol – note that the goal of our notion is different, as we are interested in what happens when *all* parties are corrupted. The notion of public verifiability has been studied for voting protocols, see e.g. [19] but to the best of our knowledge it has not been studied in the setting of secure computation, with the exception of [11], where the author presents a general transformation that turns *universally satisfiable* protocols into instances that are *auditable* in our sense. This transformation is general and slows down the computational phase of protocols, whereas our focus lies not only on the auditability, but also efficiency of the computational phase.

2 The Concept of Auditable MPC

In the usual MPC setting, the only relevant parties are those taking part in the computation. In order to introduce auditability after the fact, we add a new party which *only performs the auditing* and does not need to participate during the offline or the online phase. That is, anyone can be the auditor. In fact, the auditor does not even need to exist when the protocol is executed. In this sense, our notion of the auditor is very different from the notion of a semi-trusted third party. Nevertheless, it can check the correctness of the computation based on a protocol trace. Thus, our guarantee really holds even in the case where everyone participating in the protocol is corrupted. We are not adding a honest party to the protocol: our guarantee is that if there exist at least one honest party in the universe who cares about the output of the computation, that party can check at any time that the output is correct.
As mentioned, we put ourselves in the client-server model, so the parties involved in an auditable MPC protocols are:

The input parties: We consider m parties $\boldsymbol{\mathcal{I}_1}, ..., \boldsymbol{\mathcal{I}_m}$ with inputs $(x_1, \ldots, x_m)$.

The computing parties: We consider n parties $\boldsymbol{\mathcal{P}}_1, ..., \boldsymbol{\mathcal{P}}_n$ that participate in the computation phase. Given a set of inputs $x_1, ..., x_m$ they compute an output $y = C(x_1, ..., x_m)$ for some circuit C over a finite field. Note that $\{\boldsymbol{\mathcal{I}_1}, ..., \boldsymbol{\mathcal{I}_m}\}$ and $\{\boldsymbol{\mathcal{P}}_1, ..., \boldsymbol{\mathcal{P}}_n\}$ might not be distinct.

The protocol: The protocols starts with the input parties giving inputs to the servers – this could be done entirely via the bulletin board (if the inputs are encrypted) or using also private channels to the servers (as in the secret sharing case). Now the computing parties $\{\boldsymbol{\mathcal{P}}_1, ..., \boldsymbol{\mathcal{P}}_m\}$ interact with each other via the bulletin board (and eventually also using private channels).

The auditor: After the protocol is executed, anyone acting as the auditor $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$ can retrieve the transcript of the protocol τ from the bulletin board and (using only the circuit C and the output y) determines if the result is valid or not.

Our security notion is the standard one if there is at least one honest party (i.e. we guarantee privacy, correctness, etc.). However standard security notions do not give any guarantee in the *fully malicious* setting e.g., when all parties are corrupted. We tweak the standard notions slightly and ask an additional property, called *auditable correctness.*
This notion captures the fact that in the fully malicious case, the input can not be kept secret from $\boldsymbol{\mathcal{A}}_{DV}$. But we still want to prove if the computing parties deviate from the protocol, this will be caught by $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$, who has access to the transcript of the execution using a bulletin board $\mathcal{F}_{\text{BULLETIN}}$.
For an auditable MPC protocol and given C and $x_1, ..., x_m$ as before, we require the following:

Auditable Correctness: In the fully malicious setting, $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$ can assess the correctness of the protocol run after the fact. That is, given a protocol transcript τ the auditor $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$ outputs 'ACCEPT y' with overwhelming probability if the circuit C on input $x_1, ..., x_m$ produces the output y. At the same time the auditor $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$ will return 'REJECT' (except with negligible probability) if $\boldsymbol{\mathcal{P}}_1, ..., \boldsymbol{\mathcal{P}}_n$ deviate from the protocol such that the result is not correct.

2.1 Formal Definitions

In Figure 1 we present an ideal functionality that formalizes our notion of auditable MPC. We use the same notation as in the preceding subsection.

Functionality $\mathcal{F}_{\textsc{AuditMPC}}$

Initialize: On input (init, C) from all parties the functionality stores the circuit C and waits for the sets $A_{BI} \subseteq \{1, \ldots, m\}$ and $A_{BP} \subseteq \{1, \ldots, n\}$ from $\boldsymbol{\mathcal{A}}_{DV}$ that indicate which input and which computing parties the adversary corrupts.

Input: On input $(\mathsf{input}, \boldsymbol{\mathcal{I}_i}, \mathit{varid}, x)$ from $\boldsymbol{\mathcal{I}_i}$ and $(\mathsf{input}, \boldsymbol{\mathcal{I}_i}, \mathit{varid}, ?)$ from all parties $\boldsymbol{\mathcal{P}}_j$, with varid a fresh identifier, the functionality stores (varid, x'), where $x' = x$ if $i \notin A_{BI}$, else x' is chosen by $\boldsymbol{\mathcal{A}}_{DV}$. If $|A_{BP}| = n$, it outputs x to all parties $\boldsymbol{\mathcal{P}}_j$.

Compute: When all inputs are present, the functionality computes $y_o = C(x'_1, \ldots, x'_m)$ and:

if $|A_{BP}| = 0$ it sets $y = y_o$.

if $|A_{BP}| > 0$ it outputs y_o to $\boldsymbol{\mathcal{A}}_{DV}$ and waits for y' from $\boldsymbol{\mathcal{A}}_{DV}$. If $|A_{BP}| < n$, the functionality accepts only $y' \in \{\perp, y_o\}$. If $|A_{BP}| = n$, any value $y' \neq \perp$ is accepted. Set $y = y'$ and the flag $f = \top$ if $y' = y_o$ or $f = \perp$ otherwise.

Finally the functionality outputs y to all parties.

Audit: On input (audit) from $\boldsymbol{\mathcal{T}}_{\textsc{Audit}}$, the functionality does the following:

if $f = \top$ then output 'ACCEPT y'.

if $f = \perp \wedge y_o \neq \perp$ then output 'REJECT'.

if $y_o = \perp$ then output 'NO AUDIT POSSIBLE'.

Fig. 1: The ideal functionality that describes the online phase

The functionality resembles a standard one for MPC, but we let $\boldsymbol{\mathcal{A}}_{DV}$ replace the output of the **Output** phase if he corrupts all computing parties. In the presence of at least one honest party, $\mathcal{F}_{\textsc{AuditMPC}}$ outputs the correct value if the protocol was not aborted. Furthermore, $\boldsymbol{\mathcal{T}}_{\textsc{Audit}}$ will always be successful if the protocol was not aborted. Note that we only defined our $\mathcal{F}_{\textsc{AuditMPC}}$ for deterministic functionalities. The reason for this is that when all parties are corrupted even the auditor cannot check whether the players "followed the protocol correctly" in the sense of using real random tapes. This can be solved (using standard reductions) by letting the input parties contribute also random tapes and define the randomness used by the functionality as the XOR of those random tapes – but in the extreme case where all the input parties are corrupted this will not help us.

3 An Auditable MPC Protocol

We now present an MPC protocol that is an extension of [10,8]. We obtain a fast online phase, which almost only consists of opening shared values towards parties. Due to space restriction, we present only the online phase in the main body of this submission (using an auditable ideal functionality to implement the preprocessing) and we defer the implementation of the modified offline phase to Chapter 5.

Our setup. Let $p \in \mathbb{P}$ be a prime and G be some abelian group (in multiplicative notation) of order p where the DLP is hard to solve. Furthermore, let $g, h \in G$ be two elements such that $\langle g \rangle = \langle h \rangle = G$ and h is chosen such that $\log_g(h)$ is not known (e.g. based on some CRS). For two values $x, r \in \mathbb{Z}_p$, we define $pc(x, r) := g^x h^r$.

We assume that a secure channel towards the input parties can be established, that a broadcast functionality is available and that we have access to a bulletin board $\mathcal{F}_{\textsc{Bulletin}}$ (Fig. 2), a commitment functionality $\mathcal{F}_{\textsc{Commit}}$[2] (Fig. 3) and a random oracle $\mathcal{F}_{\textsc{ProvideRandom}}$ (Fig. 4). In practice, this functionality can be implemented in several ways, e.g. using a pseudorandom number generator and the commitment scheme $\mathcal{F}_{\textsc{Commit}}$. We use a bulletin board to keep track of all those values that are ever broadcasted (especially the commitments). Observe that no information that was posted to $\mathcal{F}_{\textsc{Bulletin}}$ can ever be changed or erased.

Note that the we inherit the $\mathcal{F}_{\textsc{Commit}}$ and $\mathcal{F}_{\textsc{ProvideRandom}}$ from the original SPDZ protocol, our only extra assumption is the existence of a bulletin board $\mathcal{F}_{\textsc{Bulletin}}$.

[2] This other commitment functionality might be implemented by a hash function, and is used whenever the linear operations of the commitment scheme are not necessary.

The ideal functionality $\mathcal{F}_{\text{Bulletin}}$

Store: On input $(\mathsf{store}, id, i, msg)$ from $\boldsymbol{\mathcal{P}}_i$, where id was not assigned yet, the functionality stores (id, i, msg).
Reveal IDs: On input (all) from party $\boldsymbol{\mathcal{P}}_i$ the functionality reveals all assigned id-values to $\boldsymbol{\mathcal{P}}_i$
Reveal message: On input (getmsg, id) from $\boldsymbol{\mathcal{P}}_i$, the functionality checks whether id was assigned already. If so, then it returns (id, j, msg) to $\boldsymbol{\mathcal{P}}_i$. Otherwise it returns $(id, \perp, \perp)$.

Fig. 2: The ideal Functionality for the Bulletin board

The ideal functionality $\mathcal{F}_{\text{Commit}}$

Commit: On input $(\mathsf{Commit}, v, r, i, \tau_v)$ by $\boldsymbol{\mathcal{P}}_i$, where both v and r are either in $\mathbb{Z}_p$ or $\perp$, and τ_v is a unique identifier, it stores (v, r, i, τ_v) on a list and outputs (i, τ_v) to all players.
Open: On input $(\mathsf{Open}, i, \tau_v)$ by $\boldsymbol{\mathcal{P}}_i$, the ideal functionality outputs (v, r, i, τ_v) to all players. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and $\boldsymbol{\mathcal{P}}_i$ is corrupt, the functionality outputs $(\perp, \perp, i, \tau_v)$ to all players.

Fig. 3: The Ideal Functionality for Commitments

Sharing values for the online phase. All computations during the online phase are done using sum-shared values. Moreover, the parties are committed to each such shared value using a MAC key α and a commitment to the shared value. The key α is also sum-shared among the parties, where party $\boldsymbol{\mathcal{P}}_i$ holds share α_i such that $\alpha = \sum_{i=1}^{n} \alpha_i$, and that the commitments to each value are publicly known.

We define the $\langle \cdot \rangle$-representation of a shared value as follows:

Definition 1. *Let $r, s, e \in \mathbb{Z}_p$, then the $\langle r \rangle$-representation of r is defined as*

$$\langle r \rangle := ((r_1, ..., r_n), (\gamma(r)_1, ..., \gamma(r)_n))$$

where $r = \sum_{i=1}^{n} r_i$ and $\alpha \cdot r = \sum_{i=1}^{n} \gamma(r)_i$. Each player $\boldsymbol{\mathcal{P}}_i$ will hold his shares $r_i, \gamma(r)_i$ of such a representation. Moreover, we define

$$\begin{aligned} \langle r \rangle + \langle s \rangle &:= ((r_1 + s_1, ..., r_n + s_n), (\gamma(r)_1 + \gamma(s)_1, ..., \gamma(r)_n + \gamma(s)_n)) \\ e \cdot \langle r \rangle &:= ((e \cdot r_1, ..., e \cdot r_n), (e \cdot \gamma(r)_1, ..., e \cdot \gamma(r)_n)) \\ e + \langle r \rangle &:= ((r_1 + e, r_2, ..., r_n), (\gamma(r)_1 + e \cdot \alpha_1, ..., \gamma(r)_n + e \cdot \alpha_n)) \end{aligned}$$

This representation is closed under linear operations:

Remark 1. Let $r, s, e \in \mathbb{Z}_p$. We say that $\langle r \rangle \hat{=} \langle s \rangle$ if both $\langle r \rangle, \langle s \rangle$ reconstruct to the same value. Then it holds that

$$\langle r \rangle + \langle s \rangle \hat{=} \langle r + s \rangle,\ e \cdot \langle r \rangle \hat{=} \langle e \cdot r \rangle,\ e + \langle r \rangle \hat{=} \langle e + r \rangle$$

During the online phase, the parties either open sharings (without revealing the MACs) or do the linear operations defined above. Together with the Beaver circuit randomization technique from [3] and a MAC checking procedure for the output phase, this already yields an actively secure MPC scheme that[3] is secure against up to $n - 1$ corrupted players.

3.1 The $[\![\cdot]\!]$-representation

In order to make SPDZ auditable we enhance the way shared values are represented and stored. In a nutshell we force the computing parties to commit to the inputs, opened values and outputs of the computation. All intermediate steps can then be checked by performing the computation using the data on $\mathcal{F}_{\text{Bulletin}}$. The

[3] Provided that the offline phase generates valid multiplication triples and random values together with MACs.

Functionality $\mathcal{F}_{\text{ProvideRandom}}$

Uniformly Random: On input $(\mathsf{urandomness}, l, p, r_i)$ from each party $\boldsymbol{\mathcal{P}}_i$, the functionality checks whether it already stored a pair $r, r_1||...||r_n, i$ in list L_2. If yes, then it outputs r to all parties. If not, then it outputs a uniformly random string $r \leftarrow \{0, \ldots, p-1\}^l$ to all players and stores $r, r_1||...||r_n$ in list L_2.

Fig. 4: The Ideal Functionality that provides random values

Procedure $\mathcal{P}_{\text{Mult}}$

Multiply$(\llbracket r \rrbracket, \llbracket s \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$**:**

(1) The players calculate $\llbracket \gamma \rrbracket = \llbracket r \rrbracket - \llbracket a \rrbracket, \llbracket \delta \rrbracket = \llbracket s \rrbracket - \llbracket b \rrbracket$
(2) The players publicly reconstruct γ, δ.
(3) Each player locally calculates $\llbracket t \rrbracket = \llbracket c \rrbracket + \delta \llbracket a \rrbracket + \gamma \llbracket b \rrbracket + \gamma\delta$
(4) Return $\llbracket t \rrbracket$ as the representation of the product.

Fig. 5: Protocol to generate the product of two $\llbracket\cdot\rrbracket$-shared values

commitment scheme is information theoretically hiding, and we will carry both the actual value $\langle r \rangle$ as well as the randomness $\langle r_{rand} \rangle$ of the commitment through the whole computation.
The commitment to a value r will be a Pedersen commitment (see [21]) $pc(r, r_{rand})$. When we open a $\llbracket\cdot\rrbracket$-representation, we reconstruct both r and r_{rand}. This way the commitment is also opened (it is already published on $\mathcal{F}_{\text{Bulletin}}$) and everyone can check that it is correct.

Definition 2. *Let $r, r_{rand} \in \mathbb{Z}_p$ and $g, h \in G$ where both g, h generate the group, then we define the $\llbracket r \rrbracket$-representation for r as*

$$\llbracket r \rrbracket := (\langle r \rangle, \langle r_{rand} \rangle, pc(r, r_{rand}))$$

where $\langle r \rangle, \langle r_{rand} \rangle$ are shared among the players as before.

The reader can easily verify that linear operations of two such representations $\llbracket a \rrbracket, \llbracket b \rrbracket$ can be efficiently computed as follows:

Definition 3. *Let $a, b, a_{rand}, b_{rand}, e \in \mathbb{Z}_p$. We define the following linear operations on $\llbracket\cdot\rrbracket$-sharings:*

$$\begin{aligned} \llbracket a \rrbracket + \llbracket b \rrbracket &:= (\langle a \rangle + \langle b \rangle, \langle a_{rand} \rangle + \langle b_{rand} \rangle, pc(a, a_{rand}) \cdot pc(b, b_{rand})) \\ e \cdot \llbracket a \rrbracket &:= (e \cdot \langle a \rangle, e \cdot \langle a_{rand} \rangle, (pc(a, a_{rand}))^e) \\ e + \llbracket a \rrbracket &:= (e + \langle a \rangle, \langle a_{rand} \rangle, pc(e, 0) \cdot pc(a, a_{rand})) \end{aligned}$$

With a slight abuse in notation, we see that

Remark 2. Let $r, s, e \in \mathbb{Z}_p$. It holds that

$$\llbracket r \rrbracket + \llbracket s \rrbracket \mathrel{\hat{=}} \llbracket r + s \rrbracket,\ e \cdot \llbracket r \rrbracket \mathrel{\hat{=}} \llbracket e \cdot r \rrbracket,\ e + \llbracket r \rrbracket \mathrel{\hat{=}} \llbracket e + r \rrbracket$$

In order to multiply two representations, we rely on [3]: Let $\llbracket r \rrbracket, \llbracket s \rrbracket$ be two values where we want to calculate a representation $\llbracket t \rrbracket$ such that $t = r \cdot s$. We assume the availability of a triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ such that a, b are uniformly random and $c = a \cdot b$. To obtain $\llbracket t \rrbracket$, use the protocol in Figure 5. Correctness and privacy are straightforward.
Finally observe that, during the online phase, one does not have to perform the computations on the commitments. Instead, *only the sharings are manipulated.*

Functionality $\mathcal{F}_{\textsc{Setup}}$

Let $\odot$ be the pointwise multiplication of vector entries.

Initialize: On input (init, p, l) from all players, the functionality stores the prime p and the SIMD factor l. $\boldsymbol{\mathcal{A}}_{DV}$ chooses the set of parties $A_{BP} \subseteq \{1, \ldots, n\}$ he corrupts.

(1) Choose a $g \in G$ and $s \in \mathbb{Z}_p^*$, set $h = g^s$. Send g, h to $\boldsymbol{\mathcal{A}}_{DV}$.
(2) For all $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\alpha_i \in \mathbb{Z}_p$, while for all $i \notin A_{BP}$, the functionality chooses $\alpha_i \leftarrow \mathbb{Z}_p$ at random.
(3) Set they key $\alpha = \sum_{i=1}^{n} \alpha_i$ and send (α_i, g, h) to $\boldsymbol{\mathcal{P}}_i, i \notin A_{BP}$.
(4) Set the flag $f = \top$.

Audit: On input (Audit), return $'\textsc{reject}'$ if $f = \bot$ or if **Initialize** or **Compute** was not executed. Else return $'\textsc{accept}'$.

Compute: On input $(\mathsf{GenerateData}, T, \rho)$ from all players with T, ρ multiples of l:

(1) $RandomValues(T, l)$:
(1.1) For each $i \notin A_{BP}$ choose uniformly random $\boldsymbol{r}_i, \boldsymbol{s}_i \leftarrow \mathbb{Z}_p^T$, send these to $\boldsymbol{\mathcal{P}}_i$ and $pc(\boldsymbol{r}_i, \boldsymbol{s}_i)$ to $\boldsymbol{\mathcal{A}}_{DV}$.
(1.2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{r}_i, \boldsymbol{s}_i \in \mathbb{Z}_p^T$
(1.3) Compute $[\![\boldsymbol{r}]\!] \leftarrow \text{Bracket}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n, T)$.
(1.4) Return $([\![\boldsymbol{r}]\!])$.

(2) $Triples(\rho, l)$:
(2.1) For $i \notin A_{BP}$, the functionality samples $\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}, \boldsymbol{b}_i, \boldsymbol{b}_{rand,i} \in \mathbb{Z}_p^{\rho}$ at random, send them to $\boldsymbol{\mathcal{P}}_i$ and $pc(\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}), pc(\boldsymbol{b}_i, \boldsymbol{b}_{rand,i})$ to $\boldsymbol{\mathcal{A}}_{DV}$.
(2.2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}, \boldsymbol{b}_i, \boldsymbol{b}_{rand,i} \in \mathbb{Z}_p^{\rho}$.
(2.3) For each $i \notin A_{BP}$ sample uniformly random $\boldsymbol{o}_i \in G^{\rho}$ and send them to $\boldsymbol{\mathcal{A}}_{DV}$.
(2.4) For $i \in A_{BP}$ let $\boldsymbol{\mathcal{A}}_{DV}$ choose $\boldsymbol{c}_i, \boldsymbol{c}_{rand,i} \in \mathbb{Z}_p^{\rho}$.
(2.5) Define $\boldsymbol{a} = \sum_{j=1}^{n} \boldsymbol{a}_j, \boldsymbol{b} = \sum_{j=1}^{n} \boldsymbol{b}_j$.
(2.6) Let $j \notin A_{BP}$ be the smallest index of an honest player(if any). For all $i \notin A_{BP}, i \neq j$ choose $\boldsymbol{c}_i \in \mathbb{Z}_p^{\rho}$ unifromly at random and $\boldsymbol{c}_{rand,i} \in \mathbb{Z}_p^{\rho}$ subject to the constraint that $\boldsymbol{o}_i = pc(\boldsymbol{c}_i, \boldsymbol{c}_{rand,i})$ using s. For $\boldsymbol{\mathcal{P}}_j$ let $\boldsymbol{c}_j = \boldsymbol{a} \odot \boldsymbol{b} - \sum_{i \in [n], i \neq j} \boldsymbol{c}_i$ and $\boldsymbol{c}_{rand,j} \in \mathbb{Z}_p^{\rho}$ such that $\boldsymbol{o}_j = pc(\boldsymbol{c}_j, \boldsymbol{c}_{rand,j})$ using s. Send $\boldsymbol{c}_i, \boldsymbol{c}_{rand,i}$ to $\boldsymbol{\mathcal{P}}_i$.
(2.7) Let $\boldsymbol{c} = \sum_{i=1}^{n} \boldsymbol{c}_i$.
(2.8) Run the macros $[\![\boldsymbol{a}]\!] \leftarrow \text{Bracket}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n, \boldsymbol{a}_{rand,1}, \ldots, \boldsymbol{a}_{rand,n}, \rho)$,
$[\![\boldsymbol{b}]\!] \leftarrow \text{Bracket}(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \boldsymbol{b}_{rand,1}, \ldots, \boldsymbol{b}_{rand,n}, \rho)$,
$[\![\boldsymbol{c}]\!] \leftarrow \text{Bracket}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n, \boldsymbol{c}_{rand,1}, \ldots, \boldsymbol{c}_{rand,n}, \rho)$.
(2.9) Let $L' = \{1, ..., \rho\}$
(2.10) If $|A_{BP}| = n$ then let $\boldsymbol{\mathcal{A}}_{DV}$ input L, otherwise let $L = L'$. If $L \neq L'$ then set $f = \bot$.
(2.11) Return $([\![\boldsymbol{a}[m]]\!], [\![\boldsymbol{b}[m]]\!], [\![\boldsymbol{c}[m]]\!])_{m \in L}$.

Macro $\text{Bracket}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n, d)$**:** This macro will be run by the functionality to create $[\![\cdot]\!]$-representations.

(1) Define $\boldsymbol{r} = \sum_{i=1}^{n} \boldsymbol{r}_i, \boldsymbol{s} = \sum_{i=1}^{n} \boldsymbol{s}_i$.
(2) If $|A_{BP}| = n$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs a vector $\boldsymbol{\Delta}_c \in G^d$.
If $\boldsymbol{\Delta}_c$ contains is not the $(1, \ldots, 1)$ vector, set $f = \bot$. If $|A_{BP}| < n$ set $\boldsymbol{\Delta}_c$ to the all-ones vector.
(3) Compute $\boldsymbol{com} = pc(\boldsymbol{r}, \boldsymbol{s}) \odot \boldsymbol{\Delta}_c$.
(4) Run macro $\langle \boldsymbol{r} \rangle \leftarrow \text{Angle}(\boldsymbol{r}_1, ..., \boldsymbol{r}_n, d)$ and $\langle \boldsymbol{s} \rangle \leftarrow \text{Angle}(\boldsymbol{s}_1, ..., \boldsymbol{s}_n, d)$.
(5) Define $[\![\boldsymbol{r}]\!] = (\langle \boldsymbol{r} \rangle, \langle \boldsymbol{s} \rangle, \boldsymbol{com})$. Return $[\![\boldsymbol{r}]\!]$.

Macro $\text{Angle}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, d)$**:** This macro will be run by the functionality to create $\langle \cdot \rangle$-representations.

(1) Define $\boldsymbol{r} = \sum_{i=1}^{n} \boldsymbol{r}_i$
(2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{\gamma}_i, \boldsymbol{\Delta}_{\gamma} \in \mathbb{Z}_p^d$, and for $i \notin A_{BP}$, choose $\boldsymbol{\gamma}_i \in_R \mathbb{Z}_p^d$ at random except for $\boldsymbol{c}_j$, with j being the smallest index not in A_{BP} (if there exists one).
(3) If $|A_{BP}| < n$ set $\boldsymbol{\gamma} = \alpha \cdot \boldsymbol{r} + \boldsymbol{\Delta}_{\gamma}$ and $\boldsymbol{\gamma}_j = \boldsymbol{\gamma} - \sum_{j \neq i=1}^{n} \boldsymbol{\gamma}_i$, else set $\boldsymbol{\gamma} = \sum_{i=1}^{n} \boldsymbol{\gamma}_i$.
(4) Define $\langle \boldsymbol{r} \rangle = (\boldsymbol{r}_1, ..., \boldsymbol{r}_n, \boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_n)$. Return $\langle \boldsymbol{r} \rangle$.

Fig. 6: The ideal functionality that describes the offline phase

Functionality $\mathcal{F}_{\text{SETUP}}$, Part 1

$\odot$ denotes pointwise multiplication of vector entries.

Initialize: On input (init, p, l) from all players, the functionality stores the prime p and the SIMD factor l. $\boldsymbol{\mathcal{A}}_{DV}$ chooses the set of parties $A_{BP} \subseteq \{1, \ldots, n\}$ he corrupts.

(1) Choose a $g \in G$ and $s \in \mathbb{Z}_p^*$, set $h = g^s$. Send g, h to $\boldsymbol{\mathcal{A}}_{DV}$.
(2) For all $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\alpha_i \in \mathbb{Z}_p$, while for all $i \notin A_{BP}$, the functionality chooses $\alpha_i \leftarrow \mathbb{Z}_p$ at random.
(3) Set they key $\alpha = \sum_{i=1}^{n} \alpha_i$ and send (α_i, g, h) to $\boldsymbol{\mathcal{P}}_i, i \notin A_{BP}$.
(4) Set the flag $f = \top$.

Audit: On input (Audit), return $'$REJECT$'$ if $f = \perp$ or if **Initialize** or **Compute** was not executed. Else return $'$ACCEPT$'$.

Compute: On input $(\mathsf{GenerateData}, T, \rho)$ from all players with T, ρ multiples of l:

(1) $RandomValues(T, l)$:
(1.1) For each $i \notin A_{BP}$ choose uniformly random $\boldsymbol{r}_i, \boldsymbol{s}_i \leftarrow \mathbb{Z}_p^T$, send these to $\boldsymbol{\mathcal{P}}_i$ and $pc(\boldsymbol{r}_i, \boldsymbol{s}_i)$ to $\boldsymbol{\mathcal{A}}_{DV}$.
(1.2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{r}_i, \boldsymbol{s}_i \in \mathbb{Z}_p^T$
(1.3) Compute $\llbracket \boldsymbol{r} \rrbracket \leftarrow \text{Bracket}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n, T)$.
(1.4) Return $(\llbracket \boldsymbol{r} \rrbracket)$.

(2) $Triples(\rho, l)$:
(2.1) For $i \notin A_{BP}$, the functionality samples $\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}, \boldsymbol{b}_i, \boldsymbol{b}_{rand,i} \in \mathbb{Z}_p^\rho$ at random, send them to $\boldsymbol{\mathcal{P}}_i$ and $pc(\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}), pc(\boldsymbol{b}_i, \boldsymbol{b}_{rand,i})$ to $\boldsymbol{\mathcal{A}}_{DV}$.
(2.2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{a}_i, \boldsymbol{a}_{rand,i}, \boldsymbol{b}_i, \boldsymbol{b}_{rand,i} \in \mathbb{Z}_p^\rho$.
(2.3) For each $i \notin A_{BP}$ sample uniformly random $\boldsymbol{o}_i \in G^\rho$ and send them to $\boldsymbol{\mathcal{A}}_{DV}$.
(2.4) For $i \in A_{BP}$ let $\boldsymbol{\mathcal{A}}_{DV}$ choose $\boldsymbol{c}_i, \boldsymbol{c}_{rand,i} \in \mathbb{Z}_p^\rho$.
(2.5) Define $\boldsymbol{a} = \sum_{j=1}^{n} \boldsymbol{a}_j, \boldsymbol{b} = \sum_{j=1}^{n} \boldsymbol{b}_j$.
(2.6) Let $j \notin A_{BP}$ be the smallest index of an honest player(if any). For all $i \notin A_{BP}, i \neq j$ choose $\boldsymbol{c}_i \in \mathbb{Z}_p^\rho$ unifromly at random and $\boldsymbol{c}_{rand,i} \in \mathbb{Z}_p^\rho$ subject to the constraint that $\boldsymbol{o}_i = pc(\boldsymbol{c}_i, \boldsymbol{c}_{rand,i})$ using s. For $\boldsymbol{\mathcal{P}}_j$ let $\boldsymbol{c}_j = \boldsymbol{a} \odot \boldsymbol{b} - \sum_{i \in [n], i \neq j} \boldsymbol{c}_i$ and $\boldsymbol{c}_{rand,j} \in \mathbb{Z}_p^\rho$ such that $\boldsymbol{o}_j = pc(\boldsymbol{c}_j, \boldsymbol{c}_{rand,j})$ using s. Send $\boldsymbol{c}_i, \boldsymbol{c}_{rand,i}$ to $\boldsymbol{\mathcal{P}}_i$.
(2.7) Let $\boldsymbol{c} = \sum_{i=1}^{n} \boldsymbol{c}_i$.
(2.8) Run the macros $\llbracket \boldsymbol{a} \rrbracket \leftarrow \text{Bracket}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n, \boldsymbol{a}_{rand,1}, \ldots, \boldsymbol{a}_{rand,n}, \rho)$,
$\llbracket \boldsymbol{b} \rrbracket \leftarrow \text{Bracket}(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \boldsymbol{b}_{rand,1}, \ldots, \boldsymbol{b}_{rand,n}, \rho)$,
$\llbracket \boldsymbol{c} \rrbracket \leftarrow \text{Bracket}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n, \boldsymbol{c}_{rand,1}, \ldots, \boldsymbol{c}_{rand,n}, \rho)$.
(2.9) Let $L' = \{1, ..., \rho\}$
(2.10) If $|A_{BP}| = n$ then let $\boldsymbol{\mathcal{A}}_{DV}$ input L, otherwise let $L = L'$. If $L \neq L'$ then set $f = \perp$.
(2.11) Return $(\llbracket \boldsymbol{a}[m] \rrbracket, \llbracket \boldsymbol{b}[m] \rrbracket, \llbracket \boldsymbol{c}[m] \rrbracket)_{m \in L}$.

Fig. 7: The ideal functionality that describes the offline phase

3.2 Shared Randomness from an Offline Phase

Our online phase relies on the availability of $\llbracket \cdot \rrbracket$-representations of random values and multiplication triples. In Figure 7 and 8 we define the functionality $\mathcal{F}_{\text{SETUP}}$ that captures the behaviour of our preprocessing protocol. This is essentially an "auditable" version of the SPDZ preprocessing functionality. If all parties are corrupted, the functionality might output an incorrect result – however this can be checked by the auditor. Crucially, even if all parties are corrupted, the functionality still outputs a random public key for the commitment scheme. This allows the auditing phase to be correct (as it ensures that the even when everyone is corrupted the commitments are binding).

3.3 The Online Phase

The online phase of our protocol is presented in Figure 10. In order to create the transcript, every party puts all values it ever *sends* or *receives* onto $\mathcal{F}_{\text{BULLETIN}}$ (except for the private reconstruction of input values).

Functionality $\mathcal{F}_{\text{Setup}}$, Part 2

Macro Bracket$(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n, d)$**:** This macro will be run by the functionality to create $[\![\cdot]\!]$-representations.

(1) Define $\boldsymbol{r} = \sum_{i=1}^{n} \boldsymbol{r}_i, \boldsymbol{s} = \sum_{i=1}^{n} \boldsymbol{s}_i$.
(2) If $|A_{BP}| = n$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs a vector $\boldsymbol{\Delta}_c \in G^d$.
If $\boldsymbol{\Delta}_c$ contains is not the $(1, \ldots, 1)$ vector, set $f = \perp$. If $|A_{BP}| < n$ set $\boldsymbol{\Delta}_c$ to the all-ones vector.
(3) Compute $\boldsymbol{com} = pc(\boldsymbol{r}, \boldsymbol{s}) \odot \boldsymbol{\Delta}_c$.
(4) Run macro $\langle \boldsymbol{r} \rangle \leftarrow$ Angle$(\boldsymbol{r}_1, ..., \boldsymbol{r}_n, d)$ and $\langle \boldsymbol{s} \rangle \leftarrow$ Angle$(\boldsymbol{s}_1, ..., \boldsymbol{s}_n, d)$.
(5) Define $[\![\boldsymbol{r}]\!] = (\langle \boldsymbol{r} \rangle, \langle \boldsymbol{s} \rangle, \boldsymbol{com})$. Return $[\![\boldsymbol{r}]\!]$.

Macro Angle$(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n, d)$**:** This macro will be run by the functionality to create $\langle \cdot \rangle$-representations.

(1) Define $\boldsymbol{r} = \sum_{i=1}^{n} \boldsymbol{r}_i$
(2) For $i \in A_{BP}$, $\boldsymbol{\mathcal{A}}_{DV}$ inputs $\boldsymbol{\gamma}_i, \boldsymbol{\Delta}_\gamma \in \mathbb{Z}_p^d$, and for $i \notin A_{BP}$, choose $\boldsymbol{\gamma}_i \in_R \mathbb{Z}_p^d$ at random except for $\boldsymbol{c}_j$, with j being the smallest index not in A_{BP} (if there exists one).
(3) If $|A_{BP}| < n$ set $\boldsymbol{\gamma} = \alpha \cdot \boldsymbol{r} + \boldsymbol{\Delta}_\gamma$ and $\boldsymbol{\gamma}_j = \boldsymbol{\gamma} - \sum_{j \neq i=1}^{n} \boldsymbol{\gamma}_i$, else set $\boldsymbol{\gamma} = \sum_{i=1}^{n} \boldsymbol{\gamma}_i$.
(4) Define $\langle \boldsymbol{r} \rangle = (\boldsymbol{r}_1, ..., \boldsymbol{r}_n, \boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_n)$. Return $\langle \boldsymbol{r} \rangle$.

Fig. 8: The ideal functionality that describes the offline phase, continued

This does not break the security, because (informally speaking) this is the same information that an $\boldsymbol{\mathcal{A}}_{DV}$ receives if he corrupts $n-1$ parties. Until now, we did not show how the MACs can be checked after the computation - this can be easily done with the protocol in Figure 11.

4 Security of the Online Phase

In this section, we will prove that our new notion of auditable correctness holds for Π_{AuditMPC}. Moreover, we also formally prove that the new commitments do not interfere with the original construction of SPDZ. We start with the following Lemma from [8] (Lemma 1). It gives some insight into correctness and soundness of the MAC check. We then prove the security of the online phase in Theorem 1.

Lemma 1. *Assume that $\mathcal{P}_{\text{CheckMac}}$ is executed over the field $\mathbb{Z}_p$. The protocol $\mathcal{P}_{\text{CheckMac}}$ is correct and sound: It returns 1 if all the values v_i and their corresponding MACs are correctly computed, and rejects except with probability $2/p$ in case at least one value or MAC is not correctly computed.*

Theorem 1. *In the $\mathcal{F}_{\text{Setup}}, \mathcal{F}_{\text{Bulletin}}, \mathcal{F}_{\text{ProvideRandom}}, \mathcal{F}_{\text{Commit}}$-hybrid model, the protocol Π_{AuditMPC} implements $\mathcal{F}_{\text{AuditMPC}}$, with computational security against any static adversary corrupting all parties if the* Discrete Logarithm Problem*(DLP) is hard in the group G.*

Proof. We prove the above statement by providing a simulator $\mathcal{S}_{\text{Online}}$ (see Figure 12) which makes the ideal world and the real world execution indistinguishable.
The simulator is divided for two cases, for the honest minority and the fully malicious setting:

(1) if there is at least one honest computing runs $\mathcal{S}_{\text{Online,normal}}$ (Fig. 13).
(2) if all computing parties are corrupt, it runs $\mathcal{S}_{\text{Online,full}}$ (Fig. 14).

Observe that in the case with at least one honest party, we can simply extend the simulator from [8] to have an **audit** procedure and support for commitments.
The argument for the indistinguishability of both worlds now runs as follows: As already mentioned, the simulator does the same as Π_{AuditMPC} for **Initialize**, **Input**, **Add**, **Multiply**, only that it uses a fixed input for the honest parties during **Input**. Since all shares are uniformly random, this cannot be distinguished from the protocol execution. During the output phase, we adjust the shares of one honest party to agree with the correct output y: The simulator already has the output of the simulated computation, which is y'. Hence it can adjust the share properly, as it also knows α. Moreover, since the discrete logarithm $\log_g(h)$ is known to $\mathcal{S}_{\text{Online,normal}}$, it can also adjust the share of the randomness value y_{rand} properly. Since all shares

Protocol $\mathit{\Pi}_{\text{AuditMPC}}$

Initialize: The parties use **Initialize** and **Compute** of $\mathcal{F}_{\text{Setup}}$ to set up the MAC key α and get their share α_i as well as enough random values $\llbracket r \rrbracket$ and triples $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ to evaluate the circuit. Then the operations are performed according to the circuit structure.

Input: $\boldsymbol{\mathcal{I}_i}$ shares the input x_i. He uses a random value $\llbracket r \rrbracket$ and does the following:

(1) $\llbracket r \rrbracket$ is privately opened to $\boldsymbol{\mathcal{I}_i}$, who checks that the Pedersen commitment on $\mathcal{F}_{\text{Bulletin}}$ is correct.

(2) $\boldsymbol{\mathcal{I}_i}$ broadcasts $\epsilon = x_i - r$, which is also stored on $\mathcal{F}_{\text{Bulletin}}$.

(3) All players compute $\llbracket x_i \rrbracket = \llbracket r \rrbracket + \epsilon$

Add: Add two values $\llbracket r \rrbracket, \llbracket s \rrbracket$:

(1) Each party locally computes $\llbracket t \rrbracket = \llbracket r \rrbracket + \llbracket s \rrbracket$, but excludes the commitments from the computation.

Multiply: Multiply two values $\llbracket r \rrbracket, \llbracket s \rrbracket$, using the multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$.

(1) The parties invoke $\mathcal{P}_{\text{Mult}}.Multiply(\llbracket r \rrbracket, \llbracket s \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, but exclude the commitments from the computation.

(2) The opened values γ, δ are sent to $\mathcal{F}_{\text{Bulletin}}$.

Output: The parties want to open the output $\llbracket y \rrbracket$. Before this can be done, the MACs on all the opened values $a_1, ..., a_t$ are checked.

(1) Let $r \leftarrow \mathcal{P}_{\text{CheckMac}}.CheckOutput(a_1, \ldots, a_t, p)$. If $r \overset{?}{=} 0$ then stop.

(2) The parties open the output $\llbracket y \rrbracket$ towards $\mathcal{F}_{\text{Bulletin}}$.

(3) Let $s \leftarrow \mathcal{P}_{\text{CheckMac}}.CheckOutput(y, y_{rand}, p)$. If $s \overset{?}{=} 0$ then y or y_{rand} is not correct (Observe that at this point, the output is already correct if at most $n-1$ parties act maliciously).

Audit: We first run **Audit** for $\mathcal{F}_{\text{Setup}}$. If it returns ′ACCEPT′ then continue, otherwise return ’NO AUDIT POSSIBLE’. Now, the following part is executed if the **Output** step of the protocol instance was completed and the delivered results were correct with respect to $\mathcal{P}_{\text{CheckMac}}$. If this does not hold, return ’NO AUDIT POSSIBLE’. We follow the gates of the evaluated circuit C, in the same order. For the i-th gate, do the following:

Input: Let $\llbracket r \rrbracket$ be the opened value and *varid* be the ID of the shared value. Set $c_{varid} = pc(\epsilon, 0) \cdot c$, where c is the commitment in $\llbracket r \rrbracket$ and ϵ is the opened difference.

Add: The parties added $\llbracket r \rrbracket$ with $varid_r$ and $\llbracket s \rrbracket$ with $varid_s$ to $\llbracket t \rrbracket$ with $varid_t$. Set $c_{varid_t} = c_{varid_r} \cdot c_{varid_s}$.

Multiply: The parties multiply $\llbracket r \rrbracket$ with $varid_r$ and $\llbracket s \rrbracket$ with $varid_s$. The output has ID $varid_t$, we use the auxiliary values $\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket$ with their respective IDs. Set $c_{varid_t} = c_{varid_c} \cdot c_{varid_a}^{\delta} \cdot c_{varid_b}^{\gamma} \cdot pc(\gamma \cdot \delta, 0)$

For the output step of the original protocol, we do the following:

Output Let c be the calculated commitment for the output value $\llbracket y \rrbracket$. Check that $c \overset{?}{=} pc(y, y_{rand})$, where both the values y, y_{rand} where opened during the output phase of the computation.
If yes, then output ’ACCEPT y’, otherwise ’REJECT’.

Fig. 9: The protocol for the online phase

look uniformly random as in the protocol, they can not be distinguished. Moreover, since the commitments are information-theoretic, they do not reveal any information to the adversary. If moreover $\boldsymbol{\mathcal{A}}_{DV}$ decides to stop the execution, then $\mathcal{S}_{\text{Online,normal}}$ will forward this to the ideal functionality and $\boldsymbol{\mathcal{A}}_{DV}$ will not receive any additional information, as in the real execution.
During the **Audit** phase, we also do exactly the same as in the protocol. Note that both **Output** and **Audit** will always reveal the correct values in the simulated case (with respect to the inputs of the dishonest parties during the input phase), hence we have to show that in the real protocol, the probability that $\boldsymbol{\mathcal{A}}_{DV}$ can cheat here is negligible.

Output: There are three ways how the output can be incorrect with respect to the inputs and the calculated function, which is if a multiplication triple was not correct even though it passed the check, or if a dishonest party successfully adjusted the MACs during the computation, or it successfully cheated during the output phase. As argued in [10], the first event only happens with probability $1/p$. For the second case, a standard calculation shows that $\boldsymbol{\mathcal{A}}_{DV}$ will then be able to compute the secret MAC key α. For the third case, Lemma 1 implies that this can only happen with probability $2/p$. Since we must have p being exponential in the security parameter, the distributions are statistically indistinguishable.

Protocol $\mathit{\Pi}_{\textsc{AuditMPC}}$

Initialize: The parties use **Initialize** and **Compute** of $\mathcal{F}_{\textsc{Setup}}$ to set up the MAC key α and get their share α_i as well as enough random values $[\![r]\!]$ and triples $([\![a]\!],[\![b]\!],[\![c]\!])$ to evaluate the circuit. Then the operations are performed according to the circuit structure.

Input: $\boldsymbol{\mathcal{I}_i}$ shares the input x_i. He uses a random value $[\![r]\!]$ and does the following:

(1) $[\![r]\!]$ is privately opened to $\boldsymbol{\mathcal{I}_i}$, who checks that the Pedersen commitment on $\mathcal{F}_{\textsc{Bulletin}}$ is correct.
(2) $\boldsymbol{\mathcal{I}_i}$ broadcasts $\epsilon = x_i - r$, which is also stored on $\mathcal{F}_{\textsc{Bulletin}}$.
(3) All players compute $[\![x_i]\!] = [\![r]\!] + \epsilon$

Add: Add two values $[\![r]\!],[\![s]\!]$:

(1) Each party locally computes $[\![t]\!] = [\![r]\!] + [\![s]\!]$, but excludes the commitments from the computation.

Multiply: Multiply two values $[\![r]\!],[\![s]\!]$, using the multiplication triple $([\![a]\!],[\![b]\!],[\![c]\!])$.

(1) The parties invoke $\mathcal{P}_{\textsc{Mult}}.\mathit{Multiply}([\![r]\!],[\![s]\!],[\![a]\!],[\![b]\!],[\![c]\!])$, but exclude the commitments from the computation.
(2) The opened values γ,δ are sent to $\mathcal{F}_{\textsc{Bulletin}}$.

Output: The parties want to open the output $[\![y]\!]$. Before this can be done, the MACs on all the opened values $a_1,...,a_t$ are checked.

(1) Let $r \leftarrow \mathcal{P}_{\textsc{CheckMac}}.\mathit{CheckOutput}(a_1,\ldots,a_t,p)$. If $r \overset{?}{=} 0$ then stop.
(2) The parties open the output $[\![y]\!]$ towards $\mathcal{F}_{\textsc{Bulletin}}$.
(3) Let $s \leftarrow \mathcal{P}_{\textsc{CheckMac}}.\mathit{CheckOutput}(y,y_{rand},p)$. If $s \overset{?}{=} 0$ then y or y_{rand} is not correct (Observe that at this point, the output is already correct if at most $n-1$ parties act maliciously).

Audit: We first run **Audit** for $\mathcal{F}_{\textsc{Setup}}$. If it returns $'$ACCEPT$'$ then continue, otherwise return 'NO AUDIT POSSIBLE'. Now, the following part is executed if the **Output** step of the protocol instance was completed and the delivered results were correct with respect to $\mathcal{P}_{\textsc{CheckMac}}$. If this does not hold, return 'NO AUDIT POSSIBLE'. We follow the gates of the evaluated circuit C, in the same order. For the i-th gate, do the following:

Input: Let $[\![r]\!]$ be the opened value and varid be the ID of the shared value. Set $c_{varid} = pc(\epsilon,0)\cdot c$, where c is the commitment in $[\![r]\!]$ and ϵ is the opened difference.

Add: The parties added $[\![r]\!]$ with varid_r and $[\![s]\!]$ with varid_s to $[\![t]\!]$ with varid_t. Set $c_{varid_t} = c_{varid_r}\cdot c_{varid_s}$.

Multiply: The parties multiply $[\![r]\!]$ with varid_r and $[\![s]\!]$ with varid_s. The output has ID varid_t, we use the auxiliary values $[\![a]\!],[\![b]\!],[\![c]\!]$ with their respective IDs. Set $c_{varid_t} = c_{varid_c}\cdot c^{\delta}_{varid_a}\cdot c^{\gamma}_{varid_b}\cdot pc(\gamma\cdot\delta,0)$

For the output step of the original protocol, we do the following:

Output Let c be the calculated commitment for the output value $[\![y]\!]$. Check that $c \overset{?}{=} pc(y,y_{rand})$, where both the values y,y_{rand} where opened during the output phase of the computation.
If yes, then output 'ACCEPT y', otherwise 'REJECT'.

Fig. 10: The protocol for the online phase

Procedure $\mathcal{P}_{\textsc{CheckMac}}$

$\mathit{CheckOutput}(v_1,...,v_t,m)$ Here we check whether the MACs hold on t partially opened values.

(1) Each party samples a value r_i and sends $(\mathsf{urandomness},t,m,r_i)$ to $\mathcal{F}_{\textsc{ProvideRandom}}$ to obtain the vector $\boldsymbol{r}$.
(2) Each party computes $v = \sum_{i=1}^{t}\boldsymbol{r}[i]\cdot v_i$.
(3) Each $\boldsymbol{\mathcal{P}}_i$ computes $\gamma_i = \sum_{j=1}^{t}\boldsymbol{r}[j]\cdot\gamma(v_j)$ and $\sigma_i = \gamma_i - \alpha_i\cdot v$.
(4) Each $\boldsymbol{\mathcal{P}}_i$ commits to σ_i using $\mathcal{F}_{\textsc{Commit}}$ as c'_i.
(5) Each c'_i is opened towards all players using $\mathcal{F}_{\textsc{Commit}}$.
(6) If $\sigma = \sum_{i=1}^{n}\sigma_i$ is 0 then return 1, otherwise return 0.

Fig. 11: Procedure to check validity of MACs

Audit: We assume that all provided commitments (before the execution) are correct except with negligible probability. As they come from $\mathcal{F}_{\textsc{Setup}}$ for both the ideal world execution and the real world execution, incorrect commitments do not lead to distinguishability. In order to cheat in the real world execution of the **Audit** phase, $\boldsymbol{\mathcal{A}}_{DV}$ must be able to provide an alternative opening for the resulting commitment, which is equal to calculating the discrete log in G and which we assume can not be done in polynomial

Simulator $\mathcal{S}_{\text{Online}}$

Wait for the set of corrupted parties A_{BP} from the environment, and let n be the number of players.

If $|A_{BP}| \neq n$, then forward all incoming messages that are not from $\mathcal{S}_{\text{Online,normal}}$ to $\mathcal{S}_{\text{Online,normal}}$, and send all messages that come from $\mathcal{S}_{\text{Online,normal}}$ to the proper recipient.

If $|A_{BP}| = n$, then forward all incoming messages that are not from $\mathcal{S}_{\text{Online,full}}$ to $\mathcal{S}_{\text{Online,full}}$, and send all messages that come from $\mathcal{S}_{\text{Online,full}}$ to the proper recipient.

Fig. 12: Simulator for the online phase

Simulator $\mathcal{S}_{\text{Online,normal}}$

Observe that values of g, h are provided as a CRS by this simulator, so we actually know $s = \log_g(h)$ here.

Initialize: The simulator sets up the bulletin board $\mathcal{F}_{\text{Bulletin}}$ and afterwards runs a copy of $\mathcal{F}_{\text{Setup}}$, with which the adversary communicates through the simulator.

Input: If $\boldsymbol{\mathcal{I}_i}$ is honest, then follow the protocol for a fake input 0. If $\boldsymbol{\mathcal{I}_i}$ is dishonest, then extract the value and send it to $\mathcal{F}_{\text{AuditMPC'}}$.

Add: Follow the protocol and call *Add* of $\mathcal{F}_{\text{AuditMPC'}}$ for the respective elements.

Multiply: Follow the protocol and call *Multiply* of $\mathcal{F}_{\text{AuditMPC'}}$ for the respective elements.

Output: The simulator obtains the output y from $\mathcal{F}_{\text{AuditMPC'}}$ and generates valid shares for the honest parties as follows:

Let $\boldsymbol{\mathcal{P}}_i$ be an honest party and y' be the current output of the computation with the adversary. Let $[\![y']\!] = (\langle y' \rangle, \langle y'_{rand} \rangle, c = pc(y', y'_{rand}))$. We change the shares of $\boldsymbol{\mathcal{P}}_i$ for $\langle y' \rangle$ from y'_i to $y'_i + (y - y')$ and γ'_i to $\gamma'_i + \alpha(y - y')$ since the simulator knows α.

Observe that $s \neq 0$, so s is invertible $\mod p$, and set $y_{rand} = (y)' - y + s \cdot y'_{rand})/s \mod p$. We now set the switch the share $y_{rand,i}$ to be $y_{rand,i} + (y_{rand} - y'_{rand})$ and $\gamma_{rand,i}$ to be $\gamma_{rand,i} + \alpha \cdot (y_{rand} - y'_{rand})$.

(1) Follow the protocol to check the MACs according to step 1 of **Output** of $\mathit{\Pi}_{\text{AuditMPC}}$. If that step fails, let $\mathcal{F}_{\text{AuditMPC'}}$ deliver $\perp$ to the honest parties and stop.

(2) Send the shares of the simulated honest parties to $\mathcal{F}_{\text{Bulletin}}$. If not all malicious parties provide their shares of $[\![y]\!]$, then let $\mathcal{F}_{\text{AuditMPC'}}$ deliver $\perp$ to all honest parties and stop.

(3) Run $\mathcal{P}_{\text{CheckMac}}$ for the output. If the output y is correct, let $\mathcal{F}_{\text{AuditMPC'}}$ deliver y to all honest parties, otherwise $\perp$.

Audit: Simulate the audit process according to $\mathit{\Pi}_{\text{AuditMPC}}$ with the malicious players. Then invoke $\mathcal{F}_{\text{AuditMPC'}}$ to run **Audit** there.

Fig. 13: Simulator for honest minority

time except with negl. probability. Moreover, after the collision is found, $\boldsymbol{\mathcal{A}}_{DV}$ must already adjust the shares and MACs either during the computation or the output phase without being detected, which both only happens with probability at most $1/p$.

As both ways of cheating in the real world execution can only be done with negligible probability, the simulation is statistically indistinguishable for the honest minority setting.

Fully malicious setting. The simulator now does not have to simulate honest parties when running the protocol with the malicious parties.

The intuition behind the simulator $\mathcal{S}_{\text{Online,full}}$ is trivial - since we cannot force $\boldsymbol{\mathcal{A}}_{DV}$ to do anything particular, we leave the adversary the option to do whatever he wants during the online phase. This is due to the fact that, since $\boldsymbol{\mathcal{A}}_{DV}$ now knows both every value and the MAC key α, he can cook up transcripts for every possible output. Moreover, since we cannot guarantee privacy, no inputs must be substituted. But note that we still have to extract inputs in order to run the **Audit** phase. This is trivially possible, since we have all the inputs that a party ever receives. Note that we have this opportunity since the adversary might

Simulator $\mathcal{S}_{\textsc{Online,full}}$

Initialize: The simulator sets up the bulletin board $\mathcal{F}_{\textsc{Bulletin}}$ and afterwards runs a copy of $\mathcal{F}_{\textsc{Setup}}$, with which the adversary communicates through the simulator.

Input: If $\boldsymbol{\mathcal{I}_i}$ is honest, then ask $\mathcal{F}_{\textsc{AuditMPC}'}$ to reveal the value. If $\boldsymbol{\mathcal{I}_i}$ is dishonest, then extract the value and send it to $\mathcal{F}_{\textsc{AuditMPC}'}$.

Add: Follow the protocol and call *Add* of $\mathcal{F}_{\textsc{AuditMPC}'}$ for the respective elements.

Multiply: Follow the protocol and call *Multiply* of $\mathcal{F}_{\textsc{AuditMPC}'}$ for the respective elements.

Output:

(1) Follow the protocol to check the MACs according to step 1 of **Output** of $\Pi_{\textsc{AuditMPC}}$. If that step fails, let $\mathcal{F}_{\textsc{AuditMPC}'}$ set $y' = \perp$ and stop.

(2) Send all the shares of the parties to $\mathcal{F}_{\textsc{Bulletin}}$. If not all malicious parties provide their shares of $[\![y]\!]$, then let $\mathcal{F}_{\textsc{AuditMPC}'}$ set $y' = \perp$ and stop.

(3) Run $\mathcal{P}_{\textsc{CheckMac}}$ for the output. If the MAC on the output o is correct, let $\mathcal{F}_{\textsc{AuditMPC}'}$ set $y' = o$, otherwise $y' = \perp$.

Audit: Run the audit process according to $\Pi_{\textsc{AuditMPC}}$ with all players. Then invoke $\mathcal{F}_{\textsc{AuditMPC}'}$ to run **Audit** there and reveal the output of $\mathcal{F}_{\textsc{AuditMPC}'}$.

Fig. 14: Simulator for the fully malicious setting

arbitrarily replace these values during the computation, but is computationally bound to the correct value in the commitment, as we will explain now.

In the ideal world execution of the **Audit** phase, the adversary can announce an arbitrary value during the **Output** phase, but the simulator will reveal this as he announces the result of **Audit** from $\mathcal{F}_{\textsc{AuditMPC}'}$. In the real world, the auditable correctness is checked by the **Audit** protocol only. Distinguishing both worlds is only possible if $\boldsymbol{\mathcal{A}_{DV}}$ can make **Audit** output 'ACCEPT y' for an *incorrect* audit trail. $\boldsymbol{\mathcal{A}_{DV}}$ might either replace the input values of a party or change shares such that they fit to another value (this is also the possibility for the output phase). In the first case, since both the commitment and the difference are public, this equals finding shares for another value x' with randomness x'_{rand} that open the commitment. In the second case, $\boldsymbol{\mathcal{A}_{DV}}$ also has to announce values such that they falsely open a commitment. As we argued before, breaking the **binding** value of the commitment is the same as computing $\log_g(h)$ in G. Hence the distinguishing probability between the ideal world and the real world is negligible for every poly-time $\boldsymbol{\mathcal{A}_{DV}}$.

The overall simulator $\mathcal{S}_{\textsc{Online}}$ is then constructed as seen in Figure 12. □

5 An Implementation of the Offline Phase

In this section, we provide an implementation of $\mathcal{F}_{\textsc{Setup}}$. To begin with, observe that our data generation protocol must consist of two phases, an actual generation phase and a checking phase. During the latter one, we check the triples so they are correct during $\Pi_{\textsc{AuditMPC}}$. We moreover have to introduce a way how to perform the *audit of the offline phase.*

To implement this, a cryptosystem is required that allows a certain number of additions and multiplications of vectors of plaintexts. We will now specify the properties of such a somewhat homomorphic cryptosystem, further details can be found e.g. in [10,8].

5.1 A Suitable Cryptosystem

We define the plaintext space $\mathcal{M}$ as $\mathcal{M} = \mathbb{Z}_p^l$. This set forms a ring under the operations '+' and '$\cdot$', which we consider as component-wise applications of the field operations.

The ring $\mathcal{A}$, which is isomorphic to $\mathbb{Z}^N$ for some integer $N \in \mathbb{N}^+$, is an intermediate space. Encryption will work as a map from $\mathcal{A}$ to some additive abelian group $\mathcal{B}$, that also respects multiplication and distributivity law under certain conditions that we will describe later. The operations of $\mathcal{A}$ will also be denoted as '+','·'. Addition will be component-wise, whereas there is no restriction on how the multiplication is implemented. In order to map $\boldsymbol{m} \in \mathcal{M}$ to an element $\boldsymbol{a} \in \mathcal{A}$ and back, there exist the two functions

$$\begin{aligned} \mathit{encode} &: \mathcal{M} \to \mathcal{A} \\ \mathit{decode} &: \ \mathcal{A} \ \to \mathcal{M} \end{aligned}$$

where *encode* is injective. We want *decode* to be the inverse of *encode* (on its image) and to be structure-preserving. Moreover, *decode* has to respect the characteristic of the field and *encode* must return *short* vectors. This is formalized as follows:

(1) $\forall \boldsymbol{m} \in \mathcal{M} : \mathit{decode}(\mathit{encode}(\boldsymbol{m})) = \boldsymbol{m}$
(2) $\forall \boldsymbol{m}_1, \boldsymbol{m}_2 \in \mathcal{M} : \mathit{decode}(\mathit{encode}(\boldsymbol{m}_1) + \mathit{encode}(\boldsymbol{m}_2)) = \boldsymbol{m}_1 + \boldsymbol{m}_2$
(3) $\forall \boldsymbol{m}_1, \boldsymbol{m}_2 \in \mathcal{M} : \mathit{decode}(\mathit{encode}(\boldsymbol{m}_1) \cdot \mathit{encode}(\boldsymbol{m}_2)) = \boldsymbol{m}_1 \cdot \boldsymbol{m}_2$
(4) $\forall \boldsymbol{a} \in \mathcal{A} : \mathit{decode}(\boldsymbol{a}) = \mathit{decode}(\boldsymbol{a} \bmod \ p)$
(5) $\forall \boldsymbol{m} \in \mathcal{M} : ||\mathit{encode}(\boldsymbol{m})||_\infty \leq \tau$ with $\tau = p/2$

Algorithms. We will now specify the cryptosystem with respect to $\mathcal{M}, \mathcal{A}$ and $\mathcal{B}$. The algorithms are considered to be probabilistic polynomial time.

$\underline{\mathit{ParamGen}(1^\lambda, \mathcal{M})}$ The algorithm outputs the dimension N of the ring $\mathcal{A}$ and descriptions for *encode* and *decode* as well as a randomized algorithm D^d_ρ, an additive abelian group $\mathcal{B}$ and a set of allowable circuits C. D^d_ρ outputs vectors $\boldsymbol{r} \in \mathbb{Z}^d$ such that $Pr[||\boldsymbol{r}||_\infty \geq \rho \mid \boldsymbol{r} \leftarrow D^d_\rho] < negl(\lambda)$. $\mathcal{B}$ has the additive operation $\oplus$ and an operation $\otimes$ that is not necessarily closed, but commutative and distributive.
C is a set of allowable arithmetic *Single Instruction Multiple Data* (SIMD) circuits over $\mathbb{Z}^l_p$, the cryptosystem must be able to evaluate these circuits on ciphertexts that are generated in a certain way. The SIMD property implies that there exists a function $f \in \mathbb{Z}_p[X_1, ..., X_{n(f)}]$ such that $\widehat{f} \in C$ evaluates the function f l times on inputs in $(\mathbb{Z}_p)^{n(f)}$ in parallel.

$\underline{\mathit{Enc}_{pk}(\boldsymbol{x}, \boldsymbol{r})}$ Let $\boldsymbol{x} \in \mathcal{A}$ and $\boldsymbol{r} \in \mathbb{Z}^d$ then this algorithm creates a $g \in \mathcal{B}$ deterministically. One can also apply this function to an $\boldsymbol{m} \in \mathcal{M}$, where it is implicitly assumed that $\mathit{Enc}_{pk}(\boldsymbol{m}) = \mathit{Enc}_{pk}(\boldsymbol{x'}, \boldsymbol{r'})$ with $\boldsymbol{x'} \leftarrow \mathit{encode}(\boldsymbol{m})$ and $\boldsymbol{r'} \leftarrow D^d_\rho$.
For the ZKPOPKs, we require that Enc_{pk} is homomorphic for at least a *small* number V of correct ciphertexts. More formally: Let $\boldsymbol{x}_1, ..., \boldsymbol{x}_V \in \mathit{image}(\mathit{encode}),\ \boldsymbol{r}_1, ..., \boldsymbol{r}_V \leftarrow D^d_\rho$. Then[4] it holds that

$$\mathit{Enc}_{pk}(\boldsymbol{x}_1 + ... + \boldsymbol{x}_V, \boldsymbol{r}_1 + ... + \boldsymbol{r}_V) = \mathit{Enc}_{pk}(\boldsymbol{x}_1, \boldsymbol{r}_1) \oplus ... \oplus \mathit{Enc}_{pk}(\boldsymbol{x}_V, \boldsymbol{r}_V)$$

We think here of V being two times as large as the security parameter *sec* of the zero knowledge proof that we will present later.

$\underline{\mathit{Dec}_{sk}(g)}$ For $g \in \mathcal{B}$ this algorithm will return an $\boldsymbol{m} \in \mathcal{M} \cup \{\perp\}$.

$\underline{\mathit{KeyGen}()}$ This algorithm samples a public key/private key pair (pk, sk).

$\underline{\mathit{KeyGen}^*}$ A *meaningless public key* $\overline{pk}$ is returned. Let $(pk, sk) \leftarrow \mathit{KeyGen}()$ and $\boldsymbol{m} \in \mathcal{M}$ be arbitrary. It require that

(1) $\mathit{Enc}_{\overline{pk}}(\boldsymbol{m})$ and $\mathit{Enc}_{\overline{pk}}(\boldsymbol{0})$ are *statistically indistinguishable.*

(2) pk and $\overline{pk}$ are *computationally indistinguishable.*

[4] We let $\mathit{image}(f)$ be the function that returns the image of the function f.

Functionality $\mathcal{F}_{\text{KeyGenDec}}$

Key generation:

(1) When receiving $(\mathsf{StartKeyGen})$ from all parties, run $P \leftarrow ParamGen(1^\lambda, \mathcal{M})$.
(2) Wait for randomness r_i from every party $\boldsymbol{\mathcal{P}}_i$.
(3) Let $r = \sum_{i=1}^{n} r_i$, and compute $(pk, sk) \leftarrow KeyGen()$ using the randomness r.
(4) Generate shares sk_i for all players consistent with sk, and send (pk, sk_i) to each party $\boldsymbol{\mathcal{P}}_i$.

Distributed decryption:

(1) When receiving $(\mathsf{StartDistDec})$ from all players, check whether there exists a shared key pair (pk, sk). If not, return $\perp$.
(2) Hereafter on receiving $(\mathsf{decrypt}, \boldsymbol{c})$ for an (B_{plain}, B_{rand}, C)-admissible $\boldsymbol{c}$ from all honest players, send $\boldsymbol{c}$ and $\boldsymbol{m} \leftarrow Dec_{sk}(\boldsymbol{c})$ to the adversary. On receiving $\boldsymbol{m'}$ from the adversary, send $(\mathsf{result}, \boldsymbol{m'})$ to all players. $\boldsymbol{m}, \boldsymbol{m'}$ can both be $\perp$
(3) On receiving $(\mathsf{decrypt}, \boldsymbol{c}, \boldsymbol{\mathcal{P}}_j)$ for an admissible $\boldsymbol{c}$, if $\boldsymbol{\mathcal{P}}_j$ is corrupt, send $\boldsymbol{c}, \boldsymbol{m} \leftarrow Dec_{sk}(\boldsymbol{c})$ to the adversary. If $\boldsymbol{\mathcal{P}}_j$ is honest, send $\boldsymbol{c}$ to the adversary. On receiving $\boldsymbol{m'}$ from the adversary, if $\boldsymbol{m'} \notin \mathcal{M}$, send $\perp$ to $\boldsymbol{\mathcal{P}}_j$, if $\boldsymbol{m'} \in \mathcal{M}$, send $Dec_{sk}(\boldsymbol{c}) + \boldsymbol{m'}$ to $\boldsymbol{\mathcal{P}}_j$.

Fig. 15: The ideal functionality for distributed key generation and decryption

Correctness. Let $n(f), f \in C$ be the number of input values of f and let $\widehat{f}$ be the embedding of f into $\mathcal{B}$ where '+' is replaced by $\oplus$, '$\cdot$' by $\otimes$ and the constant $c \in \mathbb{Z}_p$ by $Enc_{pk}(encode(\boldsymbol{c}), \boldsymbol{0})$. For data vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_{n(f)}$, let $f(\boldsymbol{x}_1, ..., \boldsymbol{x}_{n(f)})$ be the SIMD application of f to this data.

To formally express that the scheme is correct if certain bounds can be proven on the size of the randomness, we say that the scheme is (B_{plain}, B_{rand}, C)-correct if

$$\begin{aligned} Pr\Big[& Dec_{sk}(\boldsymbol{c}) \neq f(decode(\boldsymbol{x}_1), ..., decode(\boldsymbol{x}_{n(f)})) \,\Big|\, \\ & P \leftarrow ParamGen(1^\lambda, \mathcal{M}), \text{ for any } sk \text{ and } (pk, sk) \leftarrow KeyGen(), \textit{ for any } f \in C, \\ & \text{any } \boldsymbol{x}_i, \boldsymbol{r}_i \text{ with } ||\boldsymbol{x}_i||_\infty \leq B_{plain}, ||\boldsymbol{r}_i||_\infty \leq B_{rand}, decode(\boldsymbol{x}_i) \in \mathcal{M}, \\ & i \in \{1, ..., n(f)\}, \boldsymbol{c}_i \leftarrow Enc_{pk}(\boldsymbol{x}_i, \boldsymbol{r}_i) \text{ and } \boldsymbol{c} \leftarrow \widehat{f}(\boldsymbol{c}_1, ..., \boldsymbol{c}_{n(f)}) \Big] < negl(\lambda) \end{aligned}$$

for a negligible function $negl(\lambda)$. If a ciphertext $\boldsymbol{c}$ can be obtained using this chain of operations described above, then $\boldsymbol{c}$ is called (B_{plain}, B_{rand}, C)-admissible.

Distributed decryption and key generation. For the implementation, we require that the cryptosystem supports distributed key generation and decryption, as captured in $\mathcal{F}_{\text{KeyGenDec}}$.
Now let $sec \in \mathbb{N}$ be a security parameter for zero knowledge proofs, then

Definition 4 (Admissible Cryptosystem). *Let C contain formulas of the form*

$$(\sum_{i=1}^{n} x_i) \cdot (\sum_{i=1}^{n} y_i) + \sum_{i=1}^{n} z_i$$

where arbitrary x_i, y_i, z_i can be zero. A cryptosystem is called admissible if it is defined by the algorithms $(ParamGen, KeyGen, KeyGen^*, Enc, Dec)$, if it is (B_{plain}, B_{rand}, C)-correct with

$$B_{plain} = N \cdot \tau \cdot sec^2 \cdot 2^{(1/2+\nu)sec}$$
$$B_{rand} = d \cdot \rho \cdot sec^2 \cdot 2^{(1/2+\nu)sec}$$

for some arbitrary constant $\nu > 0$ and if it securely implements $\mathcal{F}_{KeyGenDec}$.

One can easily see that e.g. the Ring-LWE-based BGV scheme [7] or the BGH extension of LWE-based BGV (see [6]) have the required features.

Algorithms of the SHE scheme

```
$KeyGen()$:
    (1) Sample $\boldsymbol{a} \leftarrow R_q$ and $\boldsymbol{s}, \boldsymbol{e} \leftarrow D_{\mathbb{Z}^N,\alpha}$. $\boldsymbol{s}, \boldsymbol{e}$ are rounded such that they can be seen as $\boldsymbol{s}, \boldsymbol{e} \in R_q$
    (2) Compute $\boldsymbol{b} \leftarrow (\boldsymbol{a} \cdot \boldsymbol{s}) + (\boldsymbol{e} \cdot p)$
    (3) Set $pk \leftarrow (\boldsymbol{a}, \boldsymbol{b}), sk \leftarrow \boldsymbol{s}$

$KeyGen^*()$:
    (1) Sample $\widehat{\boldsymbol{a}}, \widehat{\boldsymbol{b}} \leftarrow R_q$
    (2) Set $\widehat{pk} \leftarrow (\widehat{\boldsymbol{a}}, \widehat{\boldsymbol{b}})$

$Enc_{pk}(\boldsymbol{x}, \boldsymbol{r})$:
    (1) Check whether $\boldsymbol{m} \in \mathcal{M}$
    (2) Set $\boldsymbol{x} \leftarrow encode(\boldsymbol{m})$ and $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) = \boldsymbol{r} \in D_\rho^d$.
    (3) Compute $\boldsymbol{c}_0 = (\boldsymbol{b} \cdot \boldsymbol{v}) + (p \cdot \boldsymbol{w}) + \boldsymbol{x}, \boldsymbol{c}_1 = (\boldsymbol{a} \cdot \boldsymbol{v}) + (p \cdot \boldsymbol{u})$
    (4) Return $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{0})$.

$Dec_{sk}(\boldsymbol{c})$:
    (1) Let $sk = \boldsymbol{s}$ and $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2)$
    (2) Calculate[a] $\boldsymbol{t} = \boldsymbol{c}_0 - (\boldsymbol{s} \cdot \boldsymbol{c}_1) - (\boldsymbol{s} \cdot \boldsymbol{s} \cdot \boldsymbol{c}_2) \bmod q$.
    (3) Return $\boldsymbol{x} \leftarrow decode(\boldsymbol{t} \bmod p)$
```

[a] This is the plaintext with some *noise* in it, and one can find a bound B such that $||\boldsymbol{t}||_\infty \leq B$.

Fig. 16: Other algorithms from the SHE scheme

5.2 Concrete Encryption Algorithms

$\underline{ParamGen(1^\lambda, \mathcal{M})}$:
Let $\mathcal{M} = \mathbb{Z}_p^l$. Consider the h-th *cyclotomic polynomial* $F = \Phi_h(x)$, where $N = \varphi(h), h \in \mathbb{N}^+$. We require that $F \bmod p$ factors into $l' \geq l$ different irreducible factors.
Now we define an algebra $R_p = \mathbb{F}_p[X]/F$ and embed $\mathcal{M}$ into R_p. We furthermore define $\mathcal{A} := \mathbb{Z}^N$ and embed R_p using the minimal representatives of the coefficients of the polynomials, as values from $[-p/2, p/2)$.
It holds that $||encode(\boldsymbol{m})||_\infty \leq p/2 = \tau$. Addition in $\mathcal{A}$ naturally complies with addition in $\mathcal{M}$, whereas the multiplication in $\mathcal{A}$ can be carried out using *polynomial multiplication.*

Now pick a large prime $q \gg p$ and set $R_q = \mathbb{F}_q[X]/F$. We consider the elements from R_p as elements in R_q as well, but depending on the size of q there might be some restrictions how the operations carry over — q has to be picked depending on C (large enough such that for circuits from C, operations in R_q behave as if they are done in $R = \mathbb{Z}[X]/F$). Therefore, let *encode* from now on implicitly map to R_q, and *decode* will map from R_q after the coefficients have been reduced mod p.
If one sets the operations in R_q to be $+$, $\cdot$, then let $\mathcal{B} = R_q^3$. $\oplus$ works componentwise and we define $\otimes$ if the third component is 0:

$$\otimes : (\boldsymbol{a}_0, \boldsymbol{a}_1, 0) \otimes (\boldsymbol{b}_0, \boldsymbol{b}_1, 0) := (\boldsymbol{a}_0 \cdot \boldsymbol{b}_0, \boldsymbol{a}_1 \cdot \boldsymbol{b}_0 + \boldsymbol{a}_0 \cdot \boldsymbol{b}_1, -\boldsymbol{a}_1 \cdot \boldsymbol{b}_1)$$

Finally, the definition of D_ρ^d will be done as follows: We define the random variable $\boldsymbol{x}$ on $\mathbb{Z}_q^N$ which is sampled according to the discrete Gaussian distribution $D_{\mathbb{Z}^N,\alpha}$. This means that every coefficient of $\boldsymbol{x} \leftarrow D_{\mathbb{Z}^N,\alpha}$ is distributed according to a Gaussian distribution with mean 0 and standard deviation $\sigma = \alpha/\sqrt{2 \cdot \pi}$. Now let $d = 3N$ and set $D_\rho^d = (D_{\mathbb{Z}^N,\alpha})^3$. Even though $(D_{\mathbb{Z}^N,\alpha})^3$ does not refer to q explicitly, the elements still origin from $(\mathbb{Z}_q^N)^3$ as defined above.

Protocol $\Pi_{\textsc{DistDec}}$

Initialize: Each party received a ciphertext $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2)$ and a bound B on the norm and computes

$$\boldsymbol{u}_i = \begin{cases} \boldsymbol{c}_0 - (\boldsymbol{s}_{i,1} \cdot \boldsymbol{c}_1) - (\boldsymbol{s}_{i,2} \cdot \boldsymbol{c}_2) & \text{if } i = 1 \\ -(\boldsymbol{s}_{i,1} \cdot \boldsymbol{c}_1) - (\boldsymbol{s}_{i,2} \cdot \boldsymbol{c}_2) & \text{if } i \neq 1 \end{cases}$$

and sets $\boldsymbol{t}_i = \boldsymbol{u}_i + p \cdot \boldsymbol{r}_i$ where $\boldsymbol{r}_i \in R_q$ is uniformly random with $||\boldsymbol{r}_i||_\infty \leq 2^{sec} B/(n \cdot p)$.

Public decryption:
(1) each $\boldsymbol{\mathcal{P}}_i$ broadcasts $\boldsymbol{t}_i$
(2) all players compute $\boldsymbol{t'} = \sum_{i=1}^n \boldsymbol{t}_i$ and obtain $\boldsymbol{m'} \leftarrow \mathit{decode}(\boldsymbol{t'} \bmod p)$

Private decryption: Decryption towards player $\boldsymbol{\mathcal{P}}_j$
(1) each party $\boldsymbol{\mathcal{P}}_i$ sends $\boldsymbol{t}_i$ to $\boldsymbol{\mathcal{P}}_j$
(2) $\boldsymbol{\mathcal{P}}_j$ computes $\boldsymbol{t'} = \sum_{i=1}^n \boldsymbol{t}_i$ and obtains $\boldsymbol{m'} \leftarrow \mathit{decode}(\boldsymbol{t'} \bmod p)$

Fig. 17: A distributed decryption protocol for the SHE scheme

Functionality $\mathcal{F}_{\textsc{KeyGen}}$

(1) When receiving (StartKeyGen) from all parties, run $P \leftarrow \mathit{ParamGen}(1^\lambda, \mathcal{M})$.
(2) Wait for randomness r_i from every party $\boldsymbol{\mathcal{P}}_i$.
(3) Let $r = \sum_{i=1}^n r_i$, and compute $(pk, sk) \leftarrow \mathit{KeyGen}()$ using the randomness r.
(4) Generate shares sk_i for all players consistent with sk, and send (pk, sk_i) to each party $\boldsymbol{\mathcal{P}}_i$.

Fig. 18: The ideal functionality for distributed key generation

Key Generation and Distributed Decryption

We now want to mention again the protocols that can be used for key generation and distributed decryption in the offline phase.
For the distributed decryption, assume that there is a plaintext that the parties want to decrypt. Every player $\boldsymbol{\mathcal{P}}_i$ will receive a share $sk_i := (\boldsymbol{s}_{1,i}, \boldsymbol{s}_{2,i})$ such that $\boldsymbol{s} = \sum_{i=1}^n \boldsymbol{s}_{1,i}$, $\boldsymbol{s} \cdot \boldsymbol{s} = \sum_{i=1}^n \boldsymbol{s}_{2,i}$ using some other MPC protocol. The protocol will proceed as follows:
In order to implement $\mathcal{F}_{\textsc{KeyGenDec}}$, one moreover needs a key generation technique. The approach from [8] is only covertly secure, but can be made actively secure using ZKPoPKs (note that this blows up the parameter size). One can also generate the shares based on another MPC protocol. The functionality is as follows:

5.3 Zero Knowledge Proofs of Plaintext Knowledge

During the online phase of the protocol, we rely on the fact that if a shared value is reconstructed, the related commitment can be opened to the same value. We ensure this (and the correct generation of ciphertexts) during the offline phase using zero knowledge proofs. Given the security parameter sec, $2 \cdot sec$ ciphertexts $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2 \cdot sec} \in \mathit{image}(\mathit{Enc}_{pk}(\cdot))$ and $sec \cdot l$ group elements $d_{1,1}, ..., d_{sec,l} \in G$, we prove the following relation:

$$\begin{aligned} R_{CTC} = \{(\boldsymbol{a}, \boldsymbol{w})| \ & \boldsymbol{a} = (\boldsymbol{c}_1, ..., \boldsymbol{c}_{2 \cdot sec}, d_{1,1}, d_{2,1}, ..., d_{sec,l}, pk), \boldsymbol{w} = (\boldsymbol{x}_1, \boldsymbol{r}_1, ..., \boldsymbol{x}_{2 \cdot sec}, \boldsymbol{r}_{2 \cdot sec}) : \\ & \forall i \in \{1, ..., sec\} : \boldsymbol{c}_i = \mathit{Enc}_{pk}(\boldsymbol{x}_i, \boldsymbol{r}_i) \wedge \boldsymbol{c}_{sec+i} = \mathit{Enc}_{pk}(\boldsymbol{x}_{sec+i}, \boldsymbol{r}_{sec+i}) \wedge \\ & ||\boldsymbol{x}_i||_\infty \leq B_{plain} \wedge ||\boldsymbol{x}_{sec+i}||_\infty \leq B_{plain} \wedge \mathit{decode}(\boldsymbol{x}_i) \in \mathbb{Z}_p^l \wedge \mathit{decode}(\boldsymbol{x}_{sec+i}) \in \mathbb{Z}_p^l \wedge \\ & ||\boldsymbol{r}_i||_\infty \leq B_{rand} \wedge ||\boldsymbol{r}_{sec+i}||_\infty \leq B_{rand} \wedge \\ & (\forall j \in \{1, ..., l\} : d_{i,j} = \mathit{pc}(\mathit{decode}(\boldsymbol{x}_i)[j], \mathit{decode}(\boldsymbol{x}_{sec+i})[j]))\} \end{aligned}$$

The protocol $\mathit{\Pi}_{\mathrm{CTC}}$

(1) For $i \in \{1, ..., V\}$ the prover generates $\boldsymbol{y}_i, \boldsymbol{y}'_i \in \mathbb{Z}^l$ and $\boldsymbol{s}_i, \boldsymbol{s}'_i \in \mathbb{Z}^d$ as follows: Let $\boldsymbol{s}_i, \boldsymbol{s}'_i$ be random such that $||\boldsymbol{s}_i||_\infty, ||\boldsymbol{s}'_i||_\infty \leq 128 \cdot d \cdot \rho \cdot sec^2$. For $\boldsymbol{y}_i, \boldsymbol{y}'_i$, let $\boldsymbol{m}_i, \boldsymbol{m}'_i \in \mathbb{Z}_p^l$ be random elements and set $\boldsymbol{y}_i = \mathit{encode}(\boldsymbol{m}_i) + \boldsymbol{u}_i$, $\boldsymbol{y}'_i = \mathit{encode}(\boldsymbol{m}'_i) + \boldsymbol{u}'_i$ where both $\boldsymbol{u}_i, \boldsymbol{u}'_i$ are generated such that each entry is a uniformly random multiple of p subject to the constraint that $||\boldsymbol{y}_i||_\infty, ||\boldsymbol{y}'_i||_\infty \leq 128 \cdot N \cdot \tau \cdot sec^2$

(2) For $i \in \{1, ..., V\}$ the prover computes $\boldsymbol{a}_i \leftarrow \mathit{Enc}_{pk}(\boldsymbol{y}_i, \boldsymbol{s}_i), \boldsymbol{a}'_i \leftarrow \mathit{Enc}_{pk}(\boldsymbol{y}'_i, \boldsymbol{s}'_i)$ and $q_{i,j} \leftarrow pc(\mathit{decode}(\boldsymbol{m}_i)[j], \mathit{decode}(\boldsymbol{m}'_i)[j])$ for $j \in \{1, ..., l\}$. For $\boldsymbol{S}, \boldsymbol{S'} \in \mathbb{Z}^{V \times d}$, he sets $\boldsymbol{S}$ to be the matrix where the ith column is $\boldsymbol{s}_i$ and $\boldsymbol{S'}$ to have $\boldsymbol{s}'_i$ as ith column respectively. Moreover, let $\boldsymbol{y} \leftarrow (y_1, ..., y_V)$, $\boldsymbol{y'} \leftarrow (y'_1, ..., y'_V), \boldsymbol{a} \leftarrow (\boldsymbol{a}_1, ..., \boldsymbol{a}_V)$ and $\boldsymbol{a'} \leftarrow (\boldsymbol{a}'_1, ..., \boldsymbol{a}'_V)$,. For the commitments, we define $\boldsymbol{q}_i \leftarrow (q_{i,1}, ..., q_{1,l})$ and $\boldsymbol{q} \leftarrow (\boldsymbol{q}_1, ..., \boldsymbol{q}_V)$.

(3) The prover sends $\boldsymbol{a}, \boldsymbol{a'}, \boldsymbol{q}$ to the verifier.

(4) The prover obtains $\boldsymbol{e}$ from $\mathcal{F}_{\text{ProvideRandom}}$ on input $(\mathsf{urandomness}, sec, 2, \boldsymbol{a}||\boldsymbol{a'}||\boldsymbol{c}||\boldsymbol{c'}||\overline{\boldsymbol{d}}||\boldsymbol{q})$.

(5) The prover sets $\boldsymbol{z} \leftarrow (\boldsymbol{z}_1, ..., \boldsymbol{z}_V), \boldsymbol{z'} \leftarrow (\boldsymbol{z}'_1, ..., \boldsymbol{z}'_V)$ where $\boldsymbol{z}^\top = \boldsymbol{y}^\top + \boldsymbol{M_e} \times \boldsymbol{x}^\top, \boldsymbol{z'}^\top = \boldsymbol{y'}^\top + \boldsymbol{M_e} \times \boldsymbol{x'}^\top$. Furthermore, he sets $\boldsymbol{T} = \boldsymbol{S} + \boldsymbol{M_e} \times \boldsymbol{R}, \boldsymbol{T'} = \boldsymbol{S'} + \boldsymbol{M_e} \times \boldsymbol{R'}$. If the ∞-norm of any value of $\boldsymbol{z}$ or $\boldsymbol{z'}$ is bigger than $128 \cdot N \cdot sec^2 - \tau \cdot sec$ or the ∞-norm of any value of $\boldsymbol{T}, \boldsymbol{T'}$ is bigger than $128 \cdot d \cdot \rho \cdot sec^2 - \rho \cdot sec$, then the protocol is restarted.
If they are smaller, then the prover sends $(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{T}, \boldsymbol{T'})$ to the verifier.

(6) The verifier obtains $\boldsymbol{e}$ from $\mathcal{F}_{\text{ProvideRandom}}$ using $(\mathsf{urandomness}, sec, 2, \boldsymbol{a}||\boldsymbol{a'}||\boldsymbol{c}||\boldsymbol{c'}||\overline{\boldsymbol{d}}||\boldsymbol{q})$. Let $\boldsymbol{t}_i$ be the ith row of $\boldsymbol{T}$ and $\boldsymbol{t}'_i$ the respective row of $\boldsymbol{T'}$. He computes $\boldsymbol{f}_i \leftarrow \mathit{Enc}_{pk}(\boldsymbol{z}_i, \boldsymbol{t}_i)$, $\boldsymbol{f}'_i \leftarrow \mathit{Enc}_{pk}(\boldsymbol{z}'_i, \boldsymbol{t}'_i)$ and sets
$\boldsymbol{f} \leftarrow (\boldsymbol{f}_1, ..., \boldsymbol{f}_V), \boldsymbol{f'} \leftarrow (\boldsymbol{f}'_1, ..., \boldsymbol{f}_V)$. In addition, the verifier computes the commitments
$g_{i,j} = pc(\mathit{decode}(\boldsymbol{z}_i)[j], \mathit{decode}(\boldsymbol{z}'_i)[j])$ for $i \in \{1, ..., V\}, j \in \{1, ..., l\}$.

(7) The verifier checks whether $\mathit{decode}(\boldsymbol{z}_i) \in \mathbb{Z}_p^l, \mathit{decode}(\boldsymbol{z}'_i) \in \mathbb{Z}_p^l$ and whether all of the following conditions hold:
(7.1) $\boldsymbol{f}^\top = \boldsymbol{a}^\top \oplus (\boldsymbol{M_e c}^\top)$
(7.2) $\boldsymbol{f'}^\top = \boldsymbol{a'}^\top \oplus (\boldsymbol{M_e c'}^\top)$
(7.3) $||\boldsymbol{z}_i||_\infty, ||\boldsymbol{z}'_i||_\infty \leq 128 \cdot N \cdot \tau \cdot sec^2$
(7.4) $||\boldsymbol{t}_i||_\infty, ||\boldsymbol{t}'_i||_\infty \leq 128 \cdot d \cdot \rho \cdot sec^2$
(7.5) Let $\boldsymbol{m}_i$ be the ith row of $\boldsymbol{M_e}$. Check that $\forall i \in \{1, ..., sec\}\ \forall j \in \{1, ..., l\} : g_{i,j} = q_{i,j} \cdot \prod_{k=1}^{sec}(d_{k,j}^{\boldsymbol{m}_i[k]})$
(7.6) If all these conditions hold, then the verifier *accepts*. Otherwise he *rejects*.

Fig. 19: The protocol for the zero knowledge proof of plaintext knowledge

To prove this statement, we execute two instances of $\mathit{\Pi}_{\mathrm{ZKPoPK}}$ from [10] for $\boldsymbol{c}_1, ..., \boldsymbol{c}_{sec}$ and $\boldsymbol{c}_{sec+1}, ..., \boldsymbol{c}_{2 \cdot sec}$ simultaneously with the same randomness. At the same time, we will put the "blinding" values of the proof into commitments, and then once again use the same randomness to prove both that we can open the commitments and that their opening values equal the plaintexts of the encryptions. We use an optimization for the proofs due to [18], called *proof with abort*, which yields smaller parameters for the cryptosystem using the zero knowledge proofs with the Fiat Shamir heuristic ([12]). Moreover, its necessary to use this heuristic to get actual randomness into the proof - since both the sender and the receiver are corrupted in the fully malicious setting, hence we have to rely on a random oracle.
For the proof, we use the same notation as SPDZ: Let $\boldsymbol{R} \in \mathbb{Z}^{sec \times d}$ be the matrix whose ith row is $\boldsymbol{r}_i$ of $\boldsymbol{c}_i$ and $\boldsymbol{R'}$ the similar matrix for $\boldsymbol{r}_{sec+i}$. Moreover, let $V = 2 \cdot sec - 1$. For a vector $\boldsymbol{e} \in \{0, 1\}^{sec}$ we define the matrix $\boldsymbol{M_e} \in \mathbb{Z}_2^{V \times sec}$ as

$$\boldsymbol{M_e}(i, j) = \begin{cases} \boldsymbol{e}_{i-j+1} & \text{if } 1 \leq i - j + 1 \leq sec \\ 0 & \text{else} \end{cases}$$

In addition, we use as abbreviations the vectors $\boldsymbol{c} \leftarrow (\boldsymbol{c}_1, ..., \boldsymbol{c}_{sec})$ and $\boldsymbol{c'} \leftarrow (\boldsymbol{c}_{sec+1}, ..., \boldsymbol{c}_{2 \cdot sec})$ for the ciphertexts. Our plaintext values will be captured in the vectors $\boldsymbol{x} \leftarrow (\boldsymbol{x}_1, ..., \boldsymbol{x}_{sec}), \boldsymbol{x'} \leftarrow (\boldsymbol{x}_{sec+1}, ..., \boldsymbol{x}_{2 \cdot sec})$ and the commitments form the list $\overline{\boldsymbol{d}} = (d_{1,1}, d_{2,1}, ..., d_{sec,l})$.
Given the protocol $\mathit{\Pi}_{\mathrm{ZKPoPK}}$, which is a honest-verifier zero knowledge proof of knowledge for a part of our relation, the following statement is straightforward:

Theorem 2. *The protocol $\mathit{\Pi}_{\mathrm{CTC}}$ is an honest-verifier zero knowledge proof of knowledge for the relation R_{CTC}.*

Procedure $\mathcal{P}_{\textsc{Reshare}}$

$\mathcal{P}_{\textsc{Reshare}}(e_{\boldsymbol{m}})$:

(1) Each $\boldsymbol{\mathcal{P}}_i$ samples a uniformly random $\boldsymbol{f}_i \in \mathbb{Z}_p^l$. We denote $\boldsymbol{f} := \sum_{j=1}^{n} \boldsymbol{f}_j$

(2) Each $\boldsymbol{\mathcal{P}}_i$ computes and broadcasts $e_{\boldsymbol{f}_i} \leftarrow Enc_{pk}(\boldsymbol{f}_i)$ to all parties and $\mathcal{F}_{\textsc{Bulletin}}$.

(3) Each $\boldsymbol{\mathcal{P}}_i$ proves with a *ZKPoPK* that $e_{\boldsymbol{f}_i}$ is (B_{plain}, B_{rand}, C)-admissible using the Random Oracle version of Π_{ZKPoPK}. It sends the proof to $\mathcal{F}_{\textsc{Bulletin}}$.

(4) The players compute $e_{\boldsymbol{f}} = \bigoplus_{i=1}^{n} e_{\boldsymbol{f}_i}$, set $e_{\boldsymbol{m}+\boldsymbol{f}} = e_{\boldsymbol{m}} \oplus e_{\boldsymbol{f}}$ and check the ZKPoPKs. If they are not correct, then they abort.

(5) The players decrypt $e_{\boldsymbol{m}+\boldsymbol{f}}$ to obtain $\boldsymbol{m} + \boldsymbol{f}$ publicly.

(6) $\boldsymbol{\mathcal{P}}_1$ sets $\boldsymbol{m}_1 = \boldsymbol{m} + \boldsymbol{f} - \boldsymbol{f}_1$ and each other player $\boldsymbol{\mathcal{P}}_i$ sets $\boldsymbol{m}_i = -\boldsymbol{f}_i$.

Fig. 20: A procedure that shares the plaintext of a publicly encrypted value

Proof. We observe that the protocol Π_{CTC} runs three instances of the SPDZ proof Π_{ZKPoPK} in parallel, two for the ciphertext vectors $\boldsymbol{c}, \boldsymbol{c'}$ and one for the commitments $\overline{\boldsymbol{d}}$. Correctness, soundness and honest-verifier zero knowledge follow directly for the first two instances due to the proof in [10], as we use different blinding values $\boldsymbol{a}, \boldsymbol{a'}$ in both instances. Hence we obtain the statements about the ciphertexts and the norms of their plaintexts, the randomness and the decodability of the plaintexts in R_{CTC}. To fill in the gaps of the proof, observe the following facts:

(1) Let us reason about the connection between the plaintext values and the commitments. First of all, we once again use an instance of the BeDOZa proof again, and we use the same randomness (observe that the group operations in the exponent coincide with the operations on the plaintexts). The connection between the plaintexts and the committed values trivially follows from the fact that our initially chosen blinding is equal, both for the ciphertexts $\boldsymbol{y}, \boldsymbol{y}'$ and the commitments $\boldsymbol{q}$. We also observe that we once again use the same randomness $\boldsymbol{M_e}$ as before, hence the operations on the ciphertexts carry over directly to the commitments.

(2) It remains to show that the commitments do not break any property of one of the other proof instances. Given two accepting proof instances for the same $\boldsymbol{a}, \boldsymbol{a'}, \boldsymbol{q}$, we refer to the fact that the cryptosystem is admissible. This means that the linear operations for the soundness proof give us plaintexts such that if we solve the similar equations for the proof of the commitments, we obtain the same values as in these plaintexts (this is because *decode* is homomorphic) except with negligible probability.

(3) For the construction of the simulator that shows the zero knowledge property, we can use the simulator for Π_{ZKPoPK} two times (for the first two instances) with the same value from the random oracle and obtain $\boldsymbol{a}, \boldsymbol{a'}, \boldsymbol{e}, \boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{T}, \boldsymbol{T'}$ that are distributed perfectly as in the real execution. This also then uniquely defines the $g_{i,j}$ for the third instance. One can now use the linearity of the scheme to obtain satisfying values $\boldsymbol{q}$ and thereby the whole transcript. The commitments $\boldsymbol{q}$ are perfectly random as in the real execution, as they cannot be related to the ciphertexts due to the IND-CPA property of the cryptosystem.

□

Observe that Π_{CTC} would be the zero knowledge proof that should be used in practice if the circuit contains many gates. For a small number of gates, the amortization technique will not pay off. We remark that the protocol Π_{CTC} can easily be adjusted to prove the relation R_{CTC} for only two ciphertexts.

5.4 Resharing Plaintexts Among Parties

This procedure shares the plaintext of a ciphertext among n parties, such that the sum of the shares equals the plaintext if all parties act honestly.

The following statements about $\mathcal{P}_{\textsc{Reshare}}$ are straightforward:

Remark 3. Assuming a (B_{plain}, B_{rand}, C)-admissible cryptosystem and $\mathcal{F}_{\textsc{Bulletin}}$, then the following statements are true about $\mathcal{P}_{\textsc{Reshare}}$ in the Random Oracle model:

Procedure $\mathcal{P}_{\textsc{ComReshare}}$

$\mathcal{P}_{\textsc{ComReshare}}(e_{\boldsymbol{m}}, e_{\boldsymbol{r},1}, ..., e_{\boldsymbol{r},n}, \boldsymbol{r}_1, ..., \boldsymbol{r}_n)$:

(1) Each $\boldsymbol{\mathcal{P}}_i$ samples a uniformly random $\boldsymbol{f}_i \in \mathbb{Z}_p^l$. We denote $\boldsymbol{f} := \sum_{j=1}^{n} \boldsymbol{f}_j$

(2) Each $\boldsymbol{\mathcal{P}}_i$ computes and broadcasts $e_{\boldsymbol{f}_i} \leftarrow Enc_{pk}(\boldsymbol{f}_i)$ to all parties and $\mathcal{F}_{\textsc{Bulletin}}$.

(3) For each $k \in \{1, \dots, l\}$, each party $\boldsymbol{\mathcal{P}}_i$ publishes $c_{f,i,k} \leftarrow pc(\boldsymbol{f}_i[k], -\boldsymbol{r}_i[k])$ on $\mathcal{F}_{\textsc{Bulletin}}$.

(4) Each $\boldsymbol{\mathcal{P}}_i$ proves with a *ZKPoPK* using Π_{CTC} that $e_{\boldsymbol{f}_i}, e_{\boldsymbol{r},i}$ are (B_{plain}, B_{rand}, C)-admissible and that the commitments hold. It sends the proof transcript to $\mathcal{F}_{\textsc{Bulletin}}$.

(5) Each player checks whether the proofs are valid.

(6) The players locally compute $e_{\boldsymbol{f}} = \bigoplus_{i=1}^{n} e_{\boldsymbol{f}_i}$ and set $e_{\boldsymbol{m}+\boldsymbol{f}} = e_{\boldsymbol{m}} \oplus e_{\boldsymbol{f}}$.

(7) The players decrypt $e_{\boldsymbol{m}+\boldsymbol{f}}$ using $\mathcal{F}_{\textsc{KeyGenDec}}$ to obtain $\boldsymbol{m} + \boldsymbol{f}$.

(8) $\boldsymbol{\mathcal{P}}_1$ sets $\boldsymbol{m}_1 = \boldsymbol{m} + \boldsymbol{f} - \boldsymbol{f}_1$ and each other player $\boldsymbol{\mathcal{P}}_i$ sets $\boldsymbol{m}_i = -\boldsymbol{f}_i$.

(9) For $k \in \{1, ..., l\}$, $\boldsymbol{\mathcal{P}}_1$ sets $c'_{m,1,k} = pc((\boldsymbol{m} + \boldsymbol{f})[k], 0)/c_{f,1,k}$ and all other players $\boldsymbol{\mathcal{P}}_i$ set $c'_{m,i,k} = c_{f,i,k}^{-1}$.

(10) All players set $e'_{\boldsymbol{m}} \leftarrow Enc_{pk}(\boldsymbol{m}+\boldsymbol{f}) \ominus (\bigoplus_{i=1}^{n} e_{\boldsymbol{f}_i})$ with the default value for the randomness of $Enc_{pk}(\boldsymbol{m}+\boldsymbol{f})$.

Fig. 21: A procedure that shares the plaintext of a publicly encrypted value together with a commitment

(1) if all parties honestly follow the protocol, then all parties afterwards obtain correct and randomly distributed shares of the plaintext of $e_{\boldsymbol{m}}$ w.h.p.

(2) if at least one and at most all parties are corrupted and the ZKPoPKs are correct, then the obtained sharing might not be a correct sharing of $\boldsymbol{m}$, but the parties know how to open all provided ciphertexts w.h.p.

The statements can be verified in [10].
In this work, we moreover need a second version of the resharing functionality. In $\mathcal{P}_{\textsc{ComReshare}}$ we will also generate commitments to the shared values.
We now give a similar characterization about $\mathcal{P}_{\textsc{ComReshare}}$ like in Remark 3:

Remark 4. Assuming a (B_{plain}, B_{rand}, C)-admissible cryptosystem,$\mathcal{F}_{\textsc{Bulletin}}$ and a group G where the DLP is hard, then the following statements are true about $\mathcal{P}_{\textsc{ComReshare}}$ in the Random Oracle model:

(1) if all parties honestly follow the protocol, then all parties afterwards obtain correct and randomly distributed shares of the plaintext of $e_{\boldsymbol{m}}$ and correct commitments to their shares (with randomness from the $e_{\boldsymbol{r},i}$) w.h.p.

(2) if at least one and at most all parties are corrupted and the ZKPoPKs are correct, then the obtained sharing might not be a correct sharing of $\boldsymbol{m}$, but the parties know how to open all provided ciphertexts and commitments w.h.p.

(3) if at least one and at most all parties are corrupted and the ZKPoPKs are correct, then $\boldsymbol{m}$ and $\boldsymbol{m}'$ might be different. The parties know a sharing of $\boldsymbol{m}'$, and $e_{\boldsymbol{m}'}$ is an admissible ciphertext and the players are committed to the values in the ciphertexts w.h.p. Moreover, the parties know how to open all provided ciphertexts and commitments w.h.p.

Observe that these statements follow from Remark 3 and Theorem 2.

5.5 Checking Correctness of Triples

We have to check that the commitments hold and that the triples are correctly formed. We rely on the standard techniques from [10].
We will now prove that, given a passed *CheckTriples* execution, the triples will have the multiplicative property whp.

Lemma 2. *Let* $D = (ParamGen, KeyGen, KeyGen^*, Enc, Dec)$ *be an admissible cryptosystem. In the* $\mathcal{F}_{\textsc{ProvideRandom}}$*-hybrid model, the test CheckTriples is correct and an adversary corrupting up to all parties can pass the test CheckTriples with non-correct triples with probability at most* $\rho/|\mathbb{Z}_p|$.

Procedure $\mathcal{P}_{\text{DataCheck}}$

CheckTriples$(t_1, ..., t_{2\rho})$**:** We put the triples into the checking and evaluation vectors $\boldsymbol{C}$ and $\boldsymbol{O}$. Then, correctness is established using the same trick as in $\mathcal{P}_{\text{CheckMac}}$. For a vector of triples $\boldsymbol{C}$, we want to access all ith $[\![\cdot]\!]$-representations in vector form as $\boldsymbol{C}(i)$.

(1) Let $\boldsymbol{C} \leftarrow (t_1, ..., t_\rho)$ and $\boldsymbol{O} \leftarrow (t_{\rho+1}, ..., t_{2\rho})$ [a]. Moreover, define $\boldsymbol{c} \leftarrow (c_1, ..., c_{2\rho})$ with $c_i \leftarrow c_{i,1}||c_{i,2}||c_{i,3}$ where $c_{i,j}$ is the commitment of the jth value of the triple t_i.
(2) Each party sends $(\mathsf{urandomness}, \rho, p, \boldsymbol{c})$ to $\mathcal{F}_{\text{ProvideRandom}}$ to generate the joint vector $\boldsymbol{t}$.
(3) Calculate $\boldsymbol{\gamma} = \boldsymbol{t} \odot \boldsymbol{O}(1) - \boldsymbol{C}(1)$ and $\boldsymbol{\Delta} = \boldsymbol{O}(2) - \boldsymbol{C}(2)$ locally.
(4) Open $\boldsymbol{\gamma}$ and $\boldsymbol{\Delta}$ towards all players.
(5) Each party evaluates $\boldsymbol{v} \leftarrow \boldsymbol{t} \odot \boldsymbol{O}(3) - \boldsymbol{C}(3) - \boldsymbol{\Delta} \odot \boldsymbol{C}(1) - \boldsymbol{\gamma} \odot \boldsymbol{C}(2) - \boldsymbol{\Delta} \odot \boldsymbol{\gamma}$ and commits to its share of $\boldsymbol{v}$ using $\mathcal{F}_{\text{Commit}}$.
(6) Each party broadcasts its opening value of the commitment to its share of $\boldsymbol{v}$.
(7) Each party locally reconstructs $\boldsymbol{v}$.
(8) For all positions i of $\boldsymbol{v}$ that are 0, output $\boldsymbol{O}[i]$ as a valid multiplication triple.

[a] Observe that one can get a lower error probability in the proof of soundness if the values are randomly assigned.

Fig. 22: A procedure to check the validity of triples

Proof. Correctness can easily be established by putting the formulas together.
Let us consider two triples $a, b, c \in \mathbb{Z}_p$ and $x, y, z \in \mathbb{Z}_p$. For $t \cdot (a \cdot b - c) = (x \cdot y - z)$ with $t \in \mathbb{Z}_p$, the following cases can happen:

(1) a, b, c **correct,** x, y, z **not:** the adversary has no chance to win
(2) a, b, c **not correct,** x, y, z **is:** the adversary can only win with probability $1/|\mathbb{Z}_p|$
(3) **both not correct:** there is only one $t \in \mathbb{Z}_p$ such that the equation holds, hence winning probability is $1/|\mathbb{Z}_p|$

If $\boldsymbol{\mathcal{A}}_{DV}$ cheats during this process and t is chosen uniformly at random, then he can cheat for every pair of triples with probability at most $1/|\mathbb{Z}_p|$ as explained above. By the union bound, this yields $\rho/|\mathbb{Z}_p|$ for *CheckTriples*. □

5.6 The Offline Phase

We define the function *diag* as $diag : \mathbb{Z}_p \to \mathcal{M},\ a \mapsto \underbrace{(a, a, ..., a)}_{l \text{ times}}$. We call such an element $Enc_{pk}(diag(a))$ a *diagonal element.* The offline phase can now be found in Figure 24 and Figure 23.

6 Security Proof of the Offline Phase

In this chapter, we will give a proof of security of the offline phase.

Theorem 3. *Let* $D = (ParamGen, KeyGen, KeyGen^*, Enc, Dec)$ *be an admissible cryptosystem. Then* Π_{Setup} *implements* $\mathcal{F}_{\text{Setup}}$ *with computational security against any static adversary corrupting at most all parties in the* $(\mathcal{F}_{\text{Commit}}, \mathcal{F}_{\text{ProvideRandom}}, \mathcal{F}_{\text{Bulletin}})$*-hybrid model if the DLP is hard in the group* G*.*

Proof. Consider the simulator in Figure 25, we will now prove that Π_{Setup} is computationally indistinguishable from $\mathcal{S}_{\text{Offline}} \diamond \mathcal{F}_{\text{Setup}}$. Once again, we will have different arguments for the honest minority and the fully malicious setting. Observe that we assume in the protocol and in the simulator that we use the non-optimized version of Π_{CTC}. We presented an optimized approach earlier for reasons of efficiency, but will prove it using a version with less overhead, to simplify the proof and hence focus on the important details. We also only present a simulator for *one* round of the offline phase - the simulation of multiple rounds is straightforward.

Procedure $\mathcal{P}_{\text{DataGen}}$

This procedure generates as many random values or multiplication triples as required. Note that we do not guarantee that the triples are correct. We will check both requirements later. Denote with l the number of plaintext slots in $\mathcal{M}$ and with $e_{\boldsymbol{a}}$ an encryption of $\boldsymbol{a} \in \mathcal{M}$. Note that $e_{\boldsymbol{\alpha}}$ encrypts a ciphertext, where every plaintext item equals the MAC key α.

RandomValues(T, l)**:** The parties generate random values, together with MACs and commitments to their shares. Set $h = \lceil T/l \rceil$, then for each $j \in \{1, \ldots, h\}$ the parties do the following:

(1) Each party $\boldsymbol{\mathcal{P}}_i$ samples uniformly random $\boldsymbol{r}_i, \boldsymbol{s}_i \in \mathcal{M}$, calculates $e_{\boldsymbol{r},i} \leftarrow Enc_{pk}(\boldsymbol{r}_i), e_{\boldsymbol{s},i} \leftarrow Enc_{pk}(\boldsymbol{s}_i)$ and broadcasts $e_{\boldsymbol{r},i}, e_{\boldsymbol{s},i}$ to all players and $\mathcal{F}_{\text{Bulletin}}$.
(2) For each $k \in \{1, \ldots, l\}$, each party $\boldsymbol{\mathcal{P}}_i$ publishes $c_{r,i,k} \leftarrow pc(\boldsymbol{r}_i[k], \boldsymbol{s}_i[k])$ on $\mathcal{F}_{\text{Bulletin}}$.
(3) Each party $\boldsymbol{\mathcal{P}}_i$ invokes Π_{CTC} on $e_{\boldsymbol{r},i}, e_{\boldsymbol{s},i}, \{c_{r,i,k}\}_{k\in\{1,\ldots,l\}}$ and publishes the transcript on $\mathcal{F}_{\text{Bulletin}}$.
(4) Each party checks all the ZKPoPKs together with the commitments. If at least one transcript is not correct, they stop here.
(5) The parties locally calculate $e_{\boldsymbol{r}} = \bigoplus_i e_{\boldsymbol{r},i}$, $e_{\boldsymbol{s}} = \bigoplus_i e_{\boldsymbol{s},i}$ as well as $\{c_{r,k} = \prod_i c_{r,i,k}\}_{k\in\{1,\ldots,l\}}$.
(6) The parties locally calculate and reshare the product with the MAC key using $\gamma_{\boldsymbol{r},i} \leftarrow \mathcal{P}_{\text{Reshare}}(e_{\boldsymbol{r}} \otimes e_{\boldsymbol{\alpha}})$, $\gamma_{\boldsymbol{s},i} \leftarrow \mathcal{P}_{\text{Reshare}}(e_{\boldsymbol{s}} \otimes e_{\boldsymbol{\alpha}})$.
(7) The values $(\boldsymbol{r}_i[k], \gamma_{\boldsymbol{r},i}[k])$, $(\boldsymbol{s}_i[k], \gamma_{\boldsymbol{s},i}[k])$, $(c_{r,k})$ are now the components of $[\![\boldsymbol{r}[k]]\!]$ for $\boldsymbol{\mathcal{P}}_i$.

Triples(ρ, l)**:** The same as for *RandomValues*, but the parties additionally multiply values to generate triples. Set $h = \lceil \rho/l \rceil$, then for $j \in \{1, \ldots, h\}$ the parties do the following:

(1) Each party $\boldsymbol{\mathcal{P}}_i$ samples uniformly random $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{f}_i, \boldsymbol{g}_i, \boldsymbol{h}_i \in \mathcal{M}$, calculates $e_{\boldsymbol{a},i} \leftarrow Enc_{pk}(\boldsymbol{a}_i)$, $e_{\boldsymbol{b},i} \leftarrow Enc_{pk}(\boldsymbol{b}_i)$ as well as $e_{\boldsymbol{f},i} \leftarrow Enc_{pk}(\boldsymbol{f}_i), e_{\boldsymbol{g},i} \leftarrow Enc_{pk}(\boldsymbol{g}_i), e_{\boldsymbol{h},i} \leftarrow Enc_{pk}(\boldsymbol{h}_i)$ and broadcasts the ciphertexts to all players and $\mathcal{F}_{\text{Bulletin}}$.
(2) For each $k \in \{1, \ldots, l\}$, each party $\boldsymbol{\mathcal{P}}_i$ publishes $c_{a,i,k} \leftarrow pc(\boldsymbol{a}_i[k], \boldsymbol{f}_i[k])$ and $c_{b,i,k} \leftarrow pc(\boldsymbol{b}_i[k], \boldsymbol{g}_i[k])$ on $\mathcal{F}_{\text{Bulletin}}$.
(3) Each party $\boldsymbol{\mathcal{P}}_i$ provides a ZKPoPK for $(\boldsymbol{a}_i, \boldsymbol{f}_i, (c_{a,i,k})_{k\in\{1,\ldots,l\}})$ and $(\boldsymbol{b}_i, \boldsymbol{g}_i, (c_{b,i,k})_{k\in\{1,\ldots,l\}})$ using Π_{CTC} and sends the transcript to $\mathcal{F}_{\text{Bulletin}}$.
(4) Each party $\boldsymbol{\mathcal{P}}_i$ checks the correctness of the ZKPoPKs of all other parties. If at least one transcript is not correct, they stop here.
(5) The parties locally calculate $e_{\boldsymbol{a}} = \bigoplus_i e_{\boldsymbol{a},i}$ and $e_{\boldsymbol{b}} = \bigoplus_i e_{\boldsymbol{b},i}$.
(6) The parties compute $e_{\boldsymbol{a}\cdot\boldsymbol{b}} = e_{\boldsymbol{a}} \otimes e_{\boldsymbol{b}}$ and invoke $\mathcal{P}_{\text{ComReshare}}(e_{\boldsymbol{a}\cdot\boldsymbol{b}}, (e_{\boldsymbol{h},i})_{i\in\{1,\ldots,n\}}, (\boldsymbol{h}_i)_{i\in\{1,\ldots,n\}})$. As a result, each party $\boldsymbol{\mathcal{P}}_i$ obtains shares $\boldsymbol{c}_i$ and all parties obtain a ciphertext $e_{\boldsymbol{c}}$ such that $\boldsymbol{c} = \sum_i \boldsymbol{c}_i$.
(7) Locally compute $e_{\boldsymbol{f}} = \bigoplus_i e_{\boldsymbol{f},i}, e_{\boldsymbol{g}} = \bigoplus_i e_{\boldsymbol{g},i}$ and $e_{\boldsymbol{h}} = \bigoplus_i e_{\boldsymbol{h},i}$. The parties compute the product of $e_{\boldsymbol{\alpha}}$ with $e_{\boldsymbol{a}}, e_{\boldsymbol{b}}, e_{\boldsymbol{c}}, e_{\boldsymbol{f}}, e_{\boldsymbol{g}}, e_{\boldsymbol{h}}$ and invoke $\mathcal{P}_{\text{Reshare}}(\cdot)$ on each such product to distribute a sharing of the MAC on each such value.

Fig. 23: Procedure $\mathcal{P}_{\text{DataGen}}$ to generate both triples and random values

Fully malicious setting In the fully malicious setting, we do not have to simulate any honest party. This makes a few steps in the proof easier. Observe that we always have to catch the case that the adversary does something not according to the protocol, which means that he can be caught during **Audit**.

Our simulator basically behaves as an observer would do in the protocol, i.e. it decrypts all information and feeds it into $\mathcal{F}_{\text{Setup}}$. Hence we do not have to argue whether the simulation is perfect, but just show that the probability of the event that happens when **Audit** from $\mathcal{F}_{\text{Setup}}$ and from Π_{Setup} reveal different values is negligible.

In our protocol, the audit process will return true if all zero knowledge proofs are correct, if the triple check was done correctly and if the revealed values open the related commitments. We observe that, if one of these conditions does not hold and hence the **Audit** fails, the same happens if the Simulator $\mathcal{S}_{\text{Offline,Full}}$ is used (as we can simply check the opened values for the commitments and since the zero knowledge proofs are complete). The case of the correctness of the triples is more subtle: A triple might be marked as correct even though the multiplicative relation does not hold (see the proof of Lemma 2), but we allow $\boldsymbol{\mathcal{A}}_{DV}$ to choose a

Protocol $\Pi_{\textsc{Setup}}$

This procedure sets up the cryptosystem for the protocol. Moreover, the random data for $\Pi_{\textsc{AuditMPC}}$ is generated that is needed during execution.

Initialize: On input (init, p, l) from all parties:

(1) The parties use $\mathcal{F}_{\textsc{KeyGenDec}}$ to generate a public key pk and a shared private key sk.
(2) The parties extract the generators $g, h \in G$ from the common reference string.
(3) Each $\boldsymbol{\mathcal{P}}_i$ generates a private $\alpha_i \in \mathbb{Z}_p$. Let $\alpha = \sum_{j=1}^{n} \alpha_j$.
(4) Each $\boldsymbol{\mathcal{P}}_i$ computes and broadcasts $e_{\boldsymbol{\alpha_i}} = Enc_{pk}(diag(\alpha_i))$.
(5) Each player $\boldsymbol{\mathcal{P}}_i$ uses $\Pi_{\textsc{ZKPoPK}}$ to prove that $e_{\boldsymbol{\alpha_i}}$ is a (B_{plain}, B_{rand}, C)-admissible, diagonal element.
(6) Each player checks the zero knowledge proofs from all other parties. If one is not correct, abort.
(7) All players compute $e_{\boldsymbol{\alpha}} = \bigoplus_{i=1}^{n} e_{\boldsymbol{\alpha_i}}$.

Compute: On input $(\mathsf{GenerateData}, T, \rho)$ from all parties and if l divides T and ρ, the players execute the subprocedures of $\mathcal{P}_{\textsc{DataGen}}$. Afterwards they check the results for correctness using $\mathcal{P}_{\textsc{DataCheck}}$.

(1) $([\![r_1]\!], ..., [\![r_T]\!]) \leftarrow \mathcal{P}_{\textsc{DataGen}}.RandomValues(T, l)$
(2) $(t_1, ..., t_\rho) \leftarrow \mathcal{P}_{\textsc{DataGen}}.Triples(\rho, l)$
(3) $(v_1, ..., v_{\rho'}) \leftarrow \mathcal{P}_{\textsc{DataCheck}}.CheckTriples(t_1, \ldots, t_\rho)$
(4) Return $([\![r_1]\!], ..., [\![r_T]\!], v_1, ..., v_{\rho'})$.

Audit: If **Compute** was executed successfully, do the following together with $\mathcal{F}_{\textsc{Bulletin}}, \mathcal{F}_{\textsc{ProvideRandom}}$:

(1) Obtain all ids and messages on $\mathcal{F}_{\textsc{Bulletin}}$.
(2) For every encryption e_i and commitment c_j, check whether there exists a transcript of Π_{CTC} or Π_{ZKPoPK} that guarantees its correctness. Otherwise return $'\textsc{reject}'$.
(3) For every transcript of Π_{CTC} or Π_{ZKPoPK}, check whether the values for each instance are on $\mathcal{F}_{\textsc{Bulletin}}$. Otherwise return $'\textsc{reject}'$.
(4) Run the verifier part for each transcript of Π_{CTC}, Π_{ZKPoPK}. If the verifier rejects, return $'\textsc{reject}'$.
(5) For each value $[\![a]\!]$ that was generated with $\mathcal{P}_{\textsc{DataGen}}$, check whether its commitment can be obtained from the commitments to the shares as in $\mathcal{P}_{\textsc{DataGen}}$. If not, return $'\textsc{reject}'$.
(6) Run $\mathcal{P}_{\textsc{DataCheck}}$ on the commitments of the triples using $\mathcal{F}_{\textsc{ProvideRandom}}$. If one of the triples that were returned by **Compute** does not open to 0 in the sanity check, return $'\textsc{reject}'$.
(7) Check for every opened value r with randomness s and commitment c whether $c = pc(r, s)$. If not, return $'\textsc{reject}'$.
(8) Return $'\textsc{accept}'$.

Fig. 24: Protocol $\Pi_{\textsc{Setup}}$ that performs the preprocessing for the online phase

Simulator $\mathcal{S}_{\textsc{Offline}}$

Wait for the set of corrupted parties A_{BP} from the environment, and let n be the number of players.

If $|A_{BP}| \neq n$, then forward all incoming messages that are not from $\mathcal{S}_{\textsc{Offline,normal}}$ to $\mathcal{S}_{\textsc{Offline,normal}}$, and send all messages that come from $\mathcal{S}_{\textsc{Offline,normal}}$ to the proper recipient.

If $|A_{BP}| = n$, then forward all incoming messages that are not from $\mathcal{S}_{\textsc{Offline,full}}$ to $\mathcal{S}_{\textsc{Offline,full}}$, and send all messages that come from $\mathcal{S}_{\textsc{Offline,full}}$ to the proper recipient.

Fig. 25: Simulator for the offline phase

set of the triples that are multiplicative. If this set does not coincide with the set of correct triples, the audit will fail later on.

Conversely, if $\mathcal{S}_{\textsc{Online,full}} \diamond \mathcal{F}_{\textsc{AuditMPC}}$ returns 'REJECT' on **Audit** then this happens if the simulator set $f = \perp$. In the case of the zero knowledge proofs, this happens if either the proof was not correct or the proof was correct and the relation did not hold (which only happens with negligible probability).

Simulator $\mathcal{S}_{\text{OFFLINE,FULL}}$, Part 1

Let n be the number of players.

Initialize:

(1) Choose random generators $g, h \in G$ such that $\log_g(h)$ is known and provide a CRS compatible with the choice.
(2) The simulator sets up $\mathcal{F}_{\text{PROVIDERANDOM}}$ locally and afterwards starts a local copy of $\mathcal{F}_{\text{SETUP}}$, with which the adversary communicates via the simulator.
(3) On input (init, p, l) from all parties, the simulator sends (init, p, l) to $\mathcal{F}_{\text{SETUP}}$.
(4) The simulator runs an instance of $\mathcal{F}_{\text{KEYGENDEC}}$ to generate a public key pk and shares of a secret key sk for all parties.
(5) Wait for the shares of $e_{\boldsymbol{\alpha},i}$ from all parties $\boldsymbol{\mathcal{P}}_i$ and the respective ZKPoPKs using Π_{ZKPoPK}. If the proofs are not correct, stop the execution of **Initialize** here. Otherwise, decrypt all $e_{\boldsymbol{\alpha},i}$ to obtain α_i.
(6) Send the α_i to $\mathcal{F}_{\text{SETUP}}$ and compute $\alpha = \sum_i \alpha_i$. Moreover, compute locally $e_{\boldsymbol{\alpha}} = \bigoplus_i e_{\boldsymbol{\alpha},i}$.

Audit:

(1) Query $\mathcal{F}_{\text{SETUP}}$ with (Audit). Return the value of $\mathcal{F}_{\text{SETUP}}$ to the requesting party.

Fig. 26: Partial simulator for the offline phase, fully malicious

Hence we see that events that trigger $\mathcal{F}_{\text{AUDITMPC}}$ to return 'REJECT' if Π_{SETUP} returns 'ACCEPT y' only occur with negligible probability. The converse can not happen at all, and the distinguishing probability must be negligible as well.

At least one honest party The proof goes along the same lines as in [10], with the difference that we now have commitments in the protocol and that the triples are already checked in this offline phase. Our subsimulator can be found in Figure 28, which as in [10] makes use of the available decryption key. A key difference is that we do provide commitments to the values of the honest parties to $\boldsymbol{\mathcal{A}}_{DV}$, but observe that the commitments are i.t. hiding, and the commitments are distributed in the simulation as they are in the actual protocol (we can choose the commitments for the multiplication in advance and later on open one to the correct values using the trapdoor $\log_g(h)$).
Based on Figure 28 one can argue that a protocol transcript does computationally not reveal any information using $KeyGen^*$, rewinding of a local environment and the zero knowledge property of Π_{CTC} based on Theorem 2, Remark 3 and 4 as well as Lemma 2 (this is equivalent to the proof in [10] and is therefore omitted here). Moreover, the outcome of **Audit** is indistinguishable as a cheating $\boldsymbol{\mathcal{A}}_{DV}$ was already caught using the zero knowledge proofs during **Compute**, and **Audit** simply also computes these checks again.
We observe that the set of triples that $\mathcal{F}_{\text{SETUP}}$ outputs will be all those triples that are correct. In the protocol execution, we will instead use the values that $CheckTriples$ outputs. The statistical distance between those outputs is ρ/p as stated in Lemma 2, which is negligible for large enough p.
The security of $\mathcal{S}_{\text{OFFLINE}}$ now trivially follows. □

7 Summary and Open Problems

In this paper, we described how to formally lift MPC into a setting where all servers are malicious. We outlined how this concept can then be securely realized on top of the SPDZ protocol. Though our approach can also be implemented for other MPC protocols, we focused on SPDZ since, even as an publicly auditable scheme, it still only consists of opening shared outputs and local computations during the online phase (excluding the $2 \cdot n$ commitments and two Random Oracle calls during **Output**). We note that our protocol would also work for Boolean circuits, but this would introduce a significant slowdown. It is an interesting future direction to design an efficient auditable protocol optimized for Boolean circuits.
A second remark we want to make is concerning the preprocessing overhead. Though the offline phase of [10] can directly be extended to support the computation of the commitments (as explained), one can reduce the

Simulator $\mathcal{S}_{\text{OFFLINE,FULL}}$ Part 2

Compute:

(1) The simulator waits for the ciphertexts $e_{\boldsymbol{r},i}, e_{\boldsymbol{s},i}$, commitments $c_{r,i,k}$ and ZKPoPKs from all parties. It sets $c_{r,k} = \prod_i c_{r,i,k}$ for all $k \in \{1,...,l\}$.

(2) The simulator decrypts the ciphertexts to $\boldsymbol{r}_i \leftarrow Dec_{sk}(e_{\boldsymbol{r},i}), \boldsymbol{s}_i \leftarrow Dec_{sk}(e_{\boldsymbol{s},i})$ and sends them to $\mathcal{F}_{\text{SETUP}}$.

(3) Compute $\boldsymbol{\Delta}_r[k] = pc(\sum_i \boldsymbol{r}_i[k], \sum_i \boldsymbol{s}_i[k])/c_{r,k}$ for all $k \in \{1,...,l\}$. If the ZKPoPKS are not all correct, then let $\boldsymbol{\Delta}_r$ be random values from G.

(4) Locally compute $e_{\boldsymbol{r}} = \bigoplus_i e_{\boldsymbol{r},i}, e_{\boldsymbol{s}} = \bigoplus_i e_{\boldsymbol{s},i}$ as well as $e_{\alpha\boldsymbol{r}} = e_{\boldsymbol{\alpha}} \otimes e_{\boldsymbol{r}}$ and $e_{\alpha\boldsymbol{s}} = e_{\boldsymbol{\alpha}} \otimes e_{\boldsymbol{s}}$.

(5) Do the following for $x = \alpha\boldsymbol{r}$ and then $x = \alpha\boldsymbol{s}$:

  (5.1) Wait for $e_{\boldsymbol{f},i}$ from each party $\boldsymbol{\mathcal{P}}_i$ and the related transcripts of Π_{ZKPoPK}. If the ZKPoPKS are not all correct, then let $\boldsymbol{\Delta}_r$ be random values from G. [a]

  (5.2) Locally compute $e_{\boldsymbol{a}} = e_x \oplus \bigoplus_i e_{\boldsymbol{f},i}$.

  (5.3) Wait for the decryption $\boldsymbol{a'}$ of $e_{\boldsymbol{a}}$ using $\mathcal{F}_{\text{KEYGENDEC}}$. Then let $\boldsymbol{\gamma}_1 \leftarrow \boldsymbol{a'} - Dec_{sk}(e_{\boldsymbol{f},1})$ and $\boldsymbol{\gamma}_i \leftarrow -Dec_{sk}(e_{\boldsymbol{f},i})$ for $i \in \{2,...,n\}$.

  (5.4) Let $\gamma_x := (\boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_n)$.

(6) Send $\boldsymbol{\Delta}_r$, $\gamma_{\alpha\boldsymbol{r}}, \gamma_{\alpha\boldsymbol{s}}$ to $\mathcal{F}_{\text{SETUP}}$.

(7) The simulator waits for the ciphertexts $e_{\boldsymbol{a},i}, e_{\boldsymbol{b},i}, e_{\boldsymbol{f},i}, e_{\boldsymbol{g},i}, e_{\boldsymbol{h},i}$, commitments $c_{a,i,k}, c_{b,i,k}$ and ZKPoPKs from all parties $\boldsymbol{\mathcal{P}}_i$. If one of the proofs is not correct, then let $c_{a,i,k}, c_{b,i,k}$ be uniformly random values in G.

(8) The simulator decrypts the ciphertexts to $\boldsymbol{a}_i \leftarrow Dec_{sk}(e_{\boldsymbol{a},i}), \boldsymbol{b}_i \leftarrow Dec_{sk}(e_{\boldsymbol{b},i}), \boldsymbol{f}_i \leftarrow Dec_{sk}(e_{\boldsymbol{f},i}), \boldsymbol{g}_i \leftarrow Dec_{sk}(e_{\boldsymbol{g},i})$ and $\boldsymbol{h}_i \leftarrow Dec_{sk}(e_{\boldsymbol{h},i})$.

(9) Locally compute $e_{\boldsymbol{a}} = \bigoplus_i e_{\boldsymbol{a},i}, e_{\boldsymbol{b}} = \bigoplus_i e_{\boldsymbol{b},i}, e_{\boldsymbol{h}} = \bigoplus_i e_{\boldsymbol{h},i}$ and compute $e_{\boldsymbol{a}\cdot\boldsymbol{b}} = e_{\boldsymbol{a}} \otimes e_{\boldsymbol{b}}$.

(10) Wait for $e'_{\boldsymbol{c},i}$ from each party $\boldsymbol{\mathcal{P}}_i$, the commitments $(c_{c',i,k})_{k\in\{1,...,l\}}$ (using $-\boldsymbol{h}_i$ as randomness) and the related transcripts of Π_{CTC}. If one of the proofs is not valid, let $c_{c',i,k}$ be random values in G.

(11) Locally compute $e'_{\boldsymbol{c}} = \bigoplus_i e'_{\boldsymbol{c},i}$ and $e_{\boldsymbol{a}\cdot\boldsymbol{b}+\boldsymbol{c}} = e'_{\boldsymbol{c}} \oplus e_{\boldsymbol{a}\cdot\boldsymbol{b}}$

(12) Wait for the decryption $\boldsymbol{a'}$ of $e_{\boldsymbol{a}\cdot\boldsymbol{b}+\boldsymbol{c}}$ using $\mathcal{F}_{\text{KEYGENDEC}}$.

(13) Let $\boldsymbol{c}_1 \leftarrow \boldsymbol{a'} - Dec_{sk}(e'_{\boldsymbol{c},1})$ and $\boldsymbol{c}_i \leftarrow -Dec_{sk}(e'_{\boldsymbol{c},i})$ for $i \in \{2,...,n\}$. Moreover, set $e_{\boldsymbol{c}} = Enc_{pk}(\boldsymbol{a'}) - \bigoplus_i e'_{\boldsymbol{c},i}$ with some standard randomness.

(14) For $k \in \{1,...,l\}$ compute $c_{c,1,k} = pc(\boldsymbol{a}'[k], 0)/c_{c',1,k}$ and $c_{c,i,k} = c^{-1}_{c',i,k}$ for each $i \in \{2,...,n\}$ locally.

(15) For $z \in \{a,b,c,f,g,h\}$ do:

  (15.1) Compute $e_{\boldsymbol{y}} \leftarrow e_\alpha \otimes e_z$ locally.

  (15.2) Wait for $e_{\boldsymbol{x},i}$ from each party $\boldsymbol{\mathcal{P}}_i$ and the related transcripts of Π_{ZKPoPK}. If one of the proofs is not valid, let $c_{c,i,k}$ be random values in G.

  (15.3) Locally compute $e_{\boldsymbol{x}} = e_{\boldsymbol{y}} \oplus \bigoplus_i e_{\boldsymbol{x},i}$.

  (15.4) Wait for the decryption $\boldsymbol{x'}$ of $e_{\boldsymbol{x}}$ using $\mathcal{F}_{\text{KEYGENDEC}}$. Then let $\boldsymbol{\gamma}_{z,1} \leftarrow \boldsymbol{x'} - Dec_{sk}(e_{\boldsymbol{x},1})$ and $\boldsymbol{\gamma}_{z,i} \leftarrow -Dec_{sk}(e_{\boldsymbol{x},i})$ for $i \in \{2,...,n\}$.

(16) Send the subvectors with the indices $l/2+1, ..., l$ of $(\boldsymbol{a}_i, \boldsymbol{f}_i, \boldsymbol{b}_i, \boldsymbol{g}_i, \boldsymbol{c}_i, \boldsymbol{h}_i)_{i\in\{1,...,n\}}$ to $\mathcal{F}_{\text{SETUP}}$.

(17) Set $c_{a,k} = \prod_i c_{a,i,k}$, $c_{b,k} = \prod_i c_{b,i,k}$ and $c_{c,k} = \prod_i c_{c,i,k}$ for all $k \in \{1,...,l\}$.

(18) Compute $\boldsymbol{\Delta}_a[k] = pc(\sum_i \boldsymbol{a}_i[k], \sum_i \boldsymbol{f}_i[k])/c_{a,k}$, $\boldsymbol{\Delta}_b[k] = pc(\sum_i \boldsymbol{b}_i[k], \sum_i \boldsymbol{g}_i[k])/c_{b,k}$ and $\boldsymbol{\Delta}_c[k] = pc(\sum_i \boldsymbol{c}_i[k], \sum_i \boldsymbol{h}_i[k])/c_{c,k}$ for all $k \in \{1,...,l\}$.

(19) For $z \in \{(a,f),(b,g),(c,h)\}$ do

  (19.1) Send the subvectors with the indices $l/2+1, ..., l$ of $\boldsymbol{\Delta}_{z[1]}$, $(\boldsymbol{\gamma}_{z[1],1}, ..., \boldsymbol{\gamma}_{z[1],n})$ and $(\boldsymbol{\gamma}_{z[2],1}, ..., \boldsymbol{\gamma}_{z[2],n})$ to $\mathcal{F}_{\text{SETUP}}$.

(20) The simulator follows the procedure $\mathcal{P}_{\text{DATACHECK}}$ with all parties[b].

(21) Send the indices of the returned values from $\mathcal{P}_{\text{DATACHECK}}$ to $\mathcal{F}_{\text{SETUP}}$.

[a] This is unrelated to this particular proof, but it will make **Audit** fail as we want.

[b] I.e. it provides randomness using its version of $\mathcal{F}_{\text{PROVIDERANDOM}}$.

Fig. 27: Partial simulator for the offline phase, fully malicious, continued

Simulator $\mathcal{S}_{\text{Offline,normal}}$

Let A_{BP} be the set of corrupted players and n be the number of players.

Initialize:

(1) Choose random generators $g, h \in G$ such that $\log_g(h)$ is known and provide a CRS compatible with the choice.
(2) The simulator sets up $\mathcal{F}_{\text{ProvideRandom}}$ locally and afterwards starts a local copy of $\mathcal{F}_{\text{Setup}}$, with which the adversary communicates via the simulator.
(3) On input (init, p, l) from all parties, the simulator sends (init, p, l) to $\mathcal{F}_{\text{Setup}}$.
(4) The simulator runs an instance of $\mathcal{F}_{\text{KeyGenDec}}$ to generate a public key pk and shares of a secret key sk for all parties.
(5) Wait for the shares of $e_{\boldsymbol{\alpha},i}$ for all $\boldsymbol{\mathcal{P}}_i, i \in A_{BP}$ and the respective ZKPoPKs using Π_{ZKPoPK}. If the proofs are not correct, stop the execution of **Initialize** here. Otherwise, decrypt all $e_{\boldsymbol{\alpha},i}$ to obtain α_i and send them to $\mathcal{F}_{\text{Setup}}$.
(6) Decrypt the broadcasted values from the honest parties to obtain $\alpha_i, i \notin A_{BP}$ and compute $\alpha = \sum_i \alpha_i$ locally.

Audit:

(1) Query $\mathcal{F}_{\text{Setup}}$ with (Audit). Return the value of $\mathcal{F}_{\text{Setup}}$ to the requesting party.

Compute:

(1) *RandomValues*: The simulator behaves exactly as in the protocol, and additionally does the following:
   - in step 1, it decrypts the ciphertexts $e_{\boldsymbol{r},i}, e_{\boldsymbol{s},i}$ to obtain the vectors $\boldsymbol{r}_i, \boldsymbol{s}_i$.
   - in step 6 it calls *SReshare* for both $e_{\boldsymbol{r}}, e_{\boldsymbol{s}}$ to obtain $\boldsymbol{\Delta}_{\gamma,r}, \boldsymbol{\Delta}_{\gamma,s}, \boldsymbol{\gamma}_{r,i}, \boldsymbol{\gamma}_{s,i}$.
   - call *RandomValues* on $\mathcal{F}_{\text{Setup}}$ and send the values $\boldsymbol{r}_i, \boldsymbol{s}_i$ in step 1.1 and $\boldsymbol{\Delta}_{\gamma,r}, \boldsymbol{\Delta}_{\gamma,s}, \boldsymbol{\gamma}_{r,i}, \boldsymbol{\gamma}_{s,i}$ in step 4 for $i \in A_{BP}$ to $\mathcal{F}_{\text{Setup}}$.

(2) *Triples*: The simulator behaves exactly as in the protocol, and additionally does the following:
   - in step 2, additionally decrypt all obtained ciphertexts from step 1 to obtain $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{f}_i, \boldsymbol{g}_i, \boldsymbol{h}_i$ for all $\boldsymbol{\mathcal{P}}_i$.
   - in step 6 run *SComReshare* to obtain $\boldsymbol{c}_i$.
   - Obtain the values $\boldsymbol{\Delta}_{\gamma,z}, \boldsymbol{\gamma}_{z,i}$ for $z \in \{a, b, c, f, g, h\}$ using *SReshare* in step 7.
   - Call *Triples* on the functionality $\mathcal{F}_{\text{Setup}}$. Then construct new vectors $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i, \boldsymbol{f}_i, \boldsymbol{g}_i, \boldsymbol{h}_i$ out of the existing ones. Pick those entries that have the following properties at the index j:
     (2.1) $j \in \lceil l/2 \rceil + 1, ..., l$
     (2.2) $\boldsymbol{a}_i[j] \cdot \boldsymbol{b}_i[j] = \boldsymbol{c}_i[j]$
     For $i \in A_{BP}$ send $\boldsymbol{a}_i, \boldsymbol{f}_i, \boldsymbol{b}_i, \boldsymbol{g}_i$ in step 2.2, $\boldsymbol{c}_i, \boldsymbol{h}_i$ in step 2.4 and $\boldsymbol{\Delta}_{\gamma,a}, \boldsymbol{\Delta}_{\gamma,f}, \boldsymbol{\gamma}_{a,i}, \boldsymbol{\gamma}_{f,i}$ in first, $\boldsymbol{\Delta}_{\gamma,b}, \boldsymbol{\Delta}_{\gamma,g}\boldsymbol{\gamma}_{b,i}, \boldsymbol{\gamma}_{g,i}$ in the second and $\boldsymbol{\Delta}_{\gamma,c}, \boldsymbol{\Delta}_{\gamma,h}\boldsymbol{\gamma}_{c,i}, \boldsymbol{\gamma}_{h,i}$ in the third execution of the macro Bracket.

(3) the simulator runs *CheckTriples* with the adversary and the values of the honest parties. Announce as result the set L of values as obtained from $\mathcal{F}_{\text{Setup}}$.

SReshare: The simulator performs the same steps as in the original SPDZ simulator (i.e. in addition to the protocol):
- in step 2, it decrypts all ciphertexts $e_{\boldsymbol{f},i}$ to obtain $\boldsymbol{f}_i$
- in step 5, it obtains $(\boldsymbol{m} + \boldsymbol{f})'$ from the adversary and $(\boldsymbol{m} + \boldsymbol{f})$ using the secret key to decrypt $e_{\boldsymbol{m}+\boldsymbol{f}}$. It then sets $\boldsymbol{\Delta}_\gamma = (\boldsymbol{m} + \boldsymbol{f})' - (\boldsymbol{m} + \boldsymbol{f})$
- the simulator sets $\boldsymbol{m}_1 = (\boldsymbol{m} - \boldsymbol{f})' - \boldsymbol{f}_1$ and $\boldsymbol{m}_i = -\boldsymbol{f}_i$ for all remaining parties

SProReshare: The simulator performs the same steps as in the protocol, and in addition extracts the following information:
- in step 2 it decrypts all ciphertexts $e_{\boldsymbol{f},i}$ to obtain $\boldsymbol{f}_i$
- in step 7, it obtains $(\boldsymbol{m} + \boldsymbol{f})'$ from the adversary and $(\boldsymbol{m} + \boldsymbol{f})$ using the secret key to decrypt $e_{\boldsymbol{m}+\boldsymbol{f}}$. It then sets $\boldsymbol{\Delta}_p = (\boldsymbol{m} + \boldsymbol{f})' - (\boldsymbol{m} + \boldsymbol{f})$
- the simulator sets $\boldsymbol{m}_1 = (\boldsymbol{m} - \boldsymbol{f})' - \boldsymbol{f}_1$ and $\boldsymbol{m}_i = -\boldsymbol{f}_i$ for all remaining parties

Fig. 28: Partial simulator for the offline phase, honest minority

computational overhead that this direct approach introduces at the expense of an only moderate slowdown of the online phase as follows: Instead of computing *one commitment per value*, one could also use s pairwise distinct generators $g_1, ..., g_s \in \mathbb{Z}_p$ together with just one randomness parameter, where generator g_i is used to commit to the ith value. A representation $(a_1, ..., a_t, r, g_1^{a_1} \cdots g_t^{a_t} h^r)$ of t values in parallel is componentwise linear, and multiplications can also be done componentwise as before. We observe that the computation of a commitment with many generators can be substantially faster than computing all commitments individually, thanks to techniques like *multi-exponentiation*. This optimization, similar to [9], works for a large class of circuits. We moreover note that, in order to use this optimization, one also has to precompute *permutations between the representations*.

Finally, we leave a working implementation of our scheme as a future work. As our protocol is very similar in structure to the original SPDZ, it should be possible to implement it easily on top of the existing codebase of [8].

Acknowledgements

Ivan Damgård acknowledges support from the Danish National Research Foundation, the National Science Foundation of China (under the grant 61061130540) and also from the CFEM research center within which part of this work was performed. Carsten Baum's work was partially supported by the European Research Commission Starting Grant 279447 and the above grants. Claudio Orlandi is supported by The Danish Council for Independent Research (DFF).

References

1. William Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In *EUROCRYPT*, pages 119–135, 2001.
2. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, pages 681–698, 2012.
3. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, Berlin, Germany, 1992.
4. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2011.
5. Elette Boyle, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, and Amit Sahai. Secure computation against adaptive auxiliary information. In *CRYPTO (1)*, pages 316–334, 2013.
6. Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. Cryptology ePrint Archive, Report 2012/565, 2012.
7. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.
8. 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. Cryptology ePrint Archive, Report 2012/642, 2012.
9. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.
10. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, Berlin, Germany, 2012.
11. Sebastiaan Jacobus Antonius de Hoogh. *Design of Large Scale Applications of Secure Multiparty Computation: Secure Linear Programming.* PhD thesis, Technische Universiteit Eindhoven, 2012.
12. Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO'86*, pages 186–194. Springer, 1987.
13. Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM Conference on Computer and Communications Security*, pages 501–512, 2012.
14. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645, 2013.

15. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.
16. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008. Preliminary full version available at `http://www.cs.illinois.edu/~mmp/pub/mpc-ot.pdf`.
17. Alptekin Küpçü and Anna Lysyanskaya. Optimistic fair exchange with multiple arbiters. In *ESORICS*, pages 488–507, 2010.
18. Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *Advances in Cryptology–ASIACRYPT 2009*, pages 598–616. Springer, 2009.
19. Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *CRYPTO*, pages 373–392, 2006.
20. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer Berlin Heidelberg, 2012.
21. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, Berlin, Germany, 1992.
22. Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 32(4):169–178, 1978.
23. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

A A Generic Implementation of Auditable MPC

Until now, we only provided a specific implementation of $\mathcal{F}_{\text{AuditMPC}}$ based on the SPDZ protocol. We now want to argue that it is possible to securely implement $\mathcal{F}_{\text{AuditMPC}}$ using generic tools, namely a "strong" semi-honest OT protocol in the sense that the protocol should be secure even if the adversary tampers with the corrupted parties internal tapes (but follow the protocol honestly), and universally composable non-interactive zero-knowledge proofs of knowledge(UC-NIZKoKs) in the *CRS* model.

First of all, note that UC-NIZKoKs trivially implement an auditable functionality: If the *CRS* and the proof are posted on the bulletin board, then the auditor (i.e., anyone) can run the verifier algorithm and *double-check* the output of the verifier.

Several notions of "strong" semi-honest protocols have been used in recent works – see Remark 1 in [16] or the notion of "semi-malicious" in [5]. In all notions different requirements of security still hold when the adversary can tamper with the randomness of otherwise semi-honest parties.

In our setting, we need that the OT protocol is still *secure* even if the adversary tampers with the random tape of one of the parties, and in addition the protocol should still be *correct* even if the adversary tampers with the random tape of *both* parties. Here *security* is defined as the usual notion of indistinguishability of the joint distribution of the view of the corrupted party and the outputs of all parties (including the honest ones) between a real execution of the protocol and a simulated one. The *correctness* requirements can similarly be defined, but we only require that indistinguishability should hold w.r.t. the output of the computation.

Note that in the case where there is at least one honest party, any semi-honest protocol can be turned into one that gives full security (not only correctness) when the adversary tampers with the randomness of the corrupted parties. The transformation goes as follows: At the beginning of the protocol $\mathcal{P}_i$ receives a random string from all other parties and redefines his random tape as the xor of its original random tape and the strings obtained externally. As long as one party is honest, $\mathcal{P}_i$'s random tape will be uniformly distributed. However it is easy to see that this transformation does not work when all parties are corrupted.

Fortunately many "natural" OT protocols, such as [1], are still correct even when the adversary tampers with the randomness of all parties. Then we can construct an "auditable" GMW-protocol against active adversaries using such an OT protocol and NIZKoK.

The protocol proceeds as follows: The input parties $\boldsymbol{\mathcal{I}_1}, ..., \boldsymbol{\mathcal{I}_m}$ share their inputs using an n-out-of-n secret sharing scheme and produce commitments to all of their shares. They now publish the commitments on the bulletin board and send one share to each server $\boldsymbol{\mathcal{P}}_j$. Those commitments should be binding even if all parties (including the input parties) are corrupted. This can be achieved by using e.g. a commitment scheme where the receiver does not send any message to the sender.
Now the computing parties $\boldsymbol{\mathcal{P}}_1, ..., \boldsymbol{\mathcal{P}}_n$ engage in an execution of the GMW-protocol using the strongly-correct OT and prove that all their messages are well formed using the NIZKoK. If there is at least one honest party, this protocol can be shown to be secure following the GMW-protocol (the only step missing is the "coin-flipping into the well" but this is taken care by the fact that the OT protocol enjoys "strong" security against semi-honest corruptions). In the audit phase $\boldsymbol{\mathcal{T}}_{\text{AUDIT}}$ checks all the NIZKs on the bulletin board and accepts y if they do and rejects if any NIZK verification fails. As the OT protocol is guaranteed to be correct even when all parties use bad randomness but follow the protocol, the auditor only outputs 'ACCEPT y' if this is the correct output.