Publicly Auditable Secure Multi-Party Computation*

Carsten Baum[†], Ivan Damgård[‡], and Claudio Orlandi[§]

{cbaum, ivan, orlandi}@cs.au.dk Aarhus University, Denmark

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 participating. This is especially relevant in the *clients-servers* setting, where many clients provide input to a secure computation performed by a few servers. 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 during the computation.

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 use-case 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 show how to achieve efficient MPC with a public audit. 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 significantly compromising the performance of SPDZ i.e. our online phase has complexity approximately twice that of SPDZ.

Keywords: Efficient Multi-Party Computation, Public Verifiability, Electronic Voting.

^{*}This is the full version of the paper which was published under the same title at the 9th Conference on Security and Cryptography for Networks (SCN 2014).

[†]Partially supported by the European Research Commission Starting Grant 279447.

[‡]Supported by 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.

[§]Supported by The Danish Council for Independent Research (DFF).

1 Introduction

During the last few years MPC has evolved from a purely theoretical to a more practical tool. Several recent protocols (e.g. BeDOZa [7], TinyOT [31] and the celebrated SPDZ [18,16]) 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 phase¹. Since these protocols allow the evaluation of an arbitrary circuit over a finite field or ring, 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, the aforementioned protocols offer very good performance. In fact the computational work per player in the SPDZ protocol is comparable to the work one has to perform 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.

The traditional solution to this is to ask every player to commit to all his secret data and to prove in zeroknowledge 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 allow 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.

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 high-speed 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 public². 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 can neither 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. We require correctness and privacy when there is at least one honest party, and in addition 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 not³. We stress that the auditor does not need to be involved (or even exist!)

 $^{^{1}}$ 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.

²Note 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).

³Of course, this only holds in the case where the computation did not abort.

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 $not.^4$

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 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 (computationally) binding. To allow the parties to open their commitments, we provide them also with a sharing of the randomness r (each party already knows a share of x). It is easy to see that this new representation of values is still linear and is therefore compatible with the existing SPDZ framework. During the computation phase, the parties ignore the commitments (they are created during the offline phase, and only the openings must be sent to $\mathcal{F}_{\text{BULLETIN}}$) 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. Clearly the offline phase of SPDZ needs to be modified, in order to produce the commitments to be used by the auditor. Moreover, we have to make this preprocessing step auditable as well.

1.2 An Example Application: Low-latency Voting from MPC

Our work can be seen as a part of a recent trend in understanding how generic MPC protocols perform (in terms of efficiency) in comparison to special-purpose protocols (see [15,24] for a discussion on private-set intersection). A notable example of *special purpose* secure computation protocols are mixed-networks (mixnets), first introduced by Chaum in 1981 [13]. Here we show how our publicly auditable version of SPDZ compares favorably with mix-nets in terms of latency.

In mix-nets a number of clients submit their encrypted inputs to some servers, who jointly shuffle and decrypt the inputs in such a way that no one should be able to link the input ciphertexts with the output plaintexts, if at least one of the shuffling servers is honest. Mix-nets are of prime importance in electronic voting (like e.g. the famous *Helios* [1] system). A disadvantage of mix-nets is that they are *inherently sequential*: server *i* cannot start shuffling before receiving the output of the shuffle performed by server i - 1. Now, given that the voter's privacy depends on the assumption that there is at least 1 out of *n* uncorrupted server, it is desirable to increase the number of parties involved in the shuffle as much as possible. However, when using mix-nets the latency of the protocol is linear in *n*, and therefore increasing *n* has a very negative impact on the total efficiency of the protocol, here measured by the time between the last voter casts his vote and the output of the election is announced. We argue here that implementing a shuffle using a generic protocol like SPDZ makes the latency independent of the number of servers performing the shuffle.

 $^{^{4}}$ In terms of feasibility, auditable MPC can be achieved by compiling a strong semi-honest protocol with NIZKs – 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 secure coin-tossing when everyone is corrupted. However, this would not lead to a very practical solution.

More formally, let n be the number of servers, m the number of input ciphertexts, λ a computational security parameter and sec a statistical security parameter. The computational latency of mix-nets, here defined as the time we have to wait before all servers have done their computational work, will be at least $O(n \cdot m \cdot \lambda^3)$.⁵ Using SPDZ, the computational latency is $O(m \cdot \log(m) \cdot sec^2)$,⁶ since the total complexity of SPDZ is linear in n and the servers work in parallel (this was even verified experimentally). Therefore mix-nets are more expensive by a factor of $(\frac{n}{\log m} \cdot \frac{\lambda^3}{sec^2})$: This is a significant speedup when n grows – note also that typical values of λ for public-key cryptography can be one or two orders of magnitudes greater than typical values for a statistical security parameter sec (only field operations are performed during the SPDZ online phase). Clearly, to verify the impact in practice one would have to implement both approaches and compare them. We leave this as an interesting future work.

The above comparison only considers the efficiency of the two approaches. However, as argued before, in applications like voting it is crucial to allow the voters to check that the outcome of the election is correct. Most mix-nets protocol already achieve public verifiability using non-interactive zero-knowledge proofs for the correctness of shuffles. This motivates our study of auditable generic protocols.

1.3 Related Work

For certain applications, there already exist *auditable protocols*. The idea is known in the context of e.g. electronic voting as *public verifiability*, and can also be found concerning online auctions and secret sharing. To the best of our knowledge, the term *public verifiability* was first used by Cohen and Fischer in [14]. Widely known publicly auditable voting protocols are those of Schoenmakers [35] and Chaum et al. [12] and the practical Helios [1]. Also stronger notions for voting protocols have been studied, see e.g. [34,29,37]. Verifiability also appeared for secret sharing schemes [35,22,36] and auctions [30,33]. We refer the reader to the mentioned papers and the references therein for more information on these subjects. It is crucial to point out that our suggested approach is not just another voting protocol – instead we lift verifiability to arbitrary secure computations. In this setting, the notion of public verifiability has not been studied, with the exception of [19], 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 approach is tailored for fast computations.

In publicly verifiable delegation of computation (see e.g. [23,21] 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 [26], but note that in our case *anyone can be the auditor* and 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 the literature. A recent work [4] investigated an enhanced 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.

2 Defining Auditable MPC

In this section, we formalize our notion of publicly auditable MPC. We add a new party to the standard formalization which *only performs the auditing* and does not need to participate during the offline or the online phase. This auditor does not even have to exist when the protocol is executed, but he can check

⁵The λ^3 factor is there because of the re-randomization step that is crucially done in every mix-net. Using "onions" of encryptions would not be more efficient.

⁶The $m \cdot \log m$ factor comes from the optimal shuffle of Ajtai et al. [3].

the correctness of a result based on a protocol transcript. This *formal hack* makes it possible to guarantee correctness even if everyone participating in the computation is corrupted⁷.

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 $\mathcal{I}_1, ..., \mathcal{I}_m$ with inputs (x_1, \ldots, x_m) .

- The computing parties: We consider *n* parties $\mathcal{P}_1, ..., \mathcal{P}_n$ 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 $\{\mathcal{I}_1, ..., \mathcal{I}_m\}$ and $\{\mathcal{P}_1, ..., \mathcal{P}_n\}$ might not be distinct.
- **The auditor:** After the protocol is executed, anyone acting as the auditor $\mathcal{T}_{\text{AUDIT}}$ can retrieve the transcript of the protocol τ from the bulletin board and (using only the circuit C and the output y) determine 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, i.e. 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 cannot be kept secret from the adversary \mathcal{A} . But we still want to prove that if the computing parties deviate from the protocol, this will be caught by $\mathcal{T}_{\text{AUDIT}}$, who has access to the transcript of the execution using a bulletin board $\mathcal{F}_{\text{BULLETIN}}$. More formally, our definition for auditable correctness is as follows:

Definition 1 (Auditable Correctness). Let C be a circuit, $x_1, ..., x_m$ be inputs to C, y be a potential output of C and τ be a protocol transcript for the evaluation of the circuit C. We say that an MPC protocol as satisfies Auditable Correctness if the following holds: The auditor \mathcal{T}_{AUDIT} with input τ 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 \mathcal{T}_{AUDIT} will return REJECT (except with negligible probability) if τ is not a transcript of an evaluation of C or if $C(x_1, ..., x_m) \neq y$.

In Figure 1 we present an ideal functionality that formalizes our notion of auditable MPC in the UC setting (where we use the same notation as before). We use this ideal world-real world paradigm, because it simplifies the proof, whereas the game-based definition gives a better intuition about auditability. The protocol/simulator transcript can then be used as in Definition 1, and a protocol that is secure in the ideal world-real world setting is also auditable correct according to the definition.

To simplify the exposition, $\mathcal{F}_{AUDITMPC}$ is only defined for one output value y. This can easily be generalized. Note that we only defined our $\mathcal{F}_{AUDITMPC}$ 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 [18,16]. We obtain a fast online phase, which almost only consists of opening shared values towards parties.

⁷We are not adding a semi-trusted third party to the actual 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.

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

- **Initialize:** On input (Init, C, p) from all parties (where C is a circuit with m inputs and one output, consisting of addition and multiplication gates over \mathbb{Z}_p):
 - (1) Wait until \mathcal{A} sends the sets $A_{BI} \subseteq \{1, \ldots, m\}$ (corrupted input parties) and $A_{BP} \subseteq \{1, \ldots, n\}$ (corrupted computing parties)
- **Input:** On input (Input, \mathcal{I}_i , vari d_x , x) from \mathcal{I}_i and (Input, \mathcal{I}_i , vari d_x ,?) from all parties \mathcal{P}_j , with vari d_x a fresh identifier:
 - (1) If $i \notin A_{BI}$ then store $(varid_x, x)$. Else let \mathcal{A} choose x' and store $(varid_x, x')$.
 - (2) If $|A_{BP}| = n$, send (Input, \mathcal{I}_i , varid, x) to all \mathcal{P}_j .

Compute: On input (Compute) from all parties \mathcal{P}_j :

- (1) If an input gate of C has no value assigned, stop here.
- (2) Compute $y_c = C(x'_1, ..., x'_m)$
- (3) if $|A_{BP}| = 0$ set $y_o = y_c$.
 - if $|A_{BP}| > 0$ output y_c to \mathcal{A} and wait for y_o from \mathcal{A} . If $|A_{BP}| < n$, the functionality accepts only $y_o \in \{\perp, y_c\}$. If $|A_{BP}| = n$, any value $y_o \in \mathbb{Z}_p \cup \{\perp\}$ is accepted.
- (4) Send (output, y_o) to all parties \mathcal{P}_j .
- Audit: On input (Audit, y) from \mathcal{T}_{AUDIT} (where $y \in \mathbb{Z}_p$), and if Compute was executed, the functionality does the following:

if $y_c = y_o = y$ then output ACCEPT y.

- if $y_o = \bot$ then output NO AUDIT POSSIBLE.
- if $y_c \neq y_o$ or $y \neq y_o$ then output REJECT.

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

Our setup. Let $p \in \mathbb{P}$ be a prime and G be some abelian group (in multiplicative notation) of order p where the *Discrete Logarithm Problem*(DLP) is hard to solve. The MPC protocol will evaluate a circuit C over \mathbb{Z}_p , whereas we use the group G to ensure auditability. Therefore, let $g, h \in G$ be two generators of the group G where 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 $\mathcal{F}_{\text{BULLETIN}}$ (Fig. 2), a commitment functionality $\mathcal{F}_{\text{COMMIT}}^{\text{COMMIT}}$ ⁸ (Fig. 3) and a procedure to jointly produce random values $\mathcal{P}_{\text{PROVIDERANDOM}}$ (Fig. 4)⁹. To implement $\mathcal{P}_{\text{PROVIDERANDOM}}$ let $\mathcal{U}_{s}(q, l)$ be a random oracle with seed $s \in \{0, 1\}^{*}$ that outputs a uniformly random element from \mathbb{Z}_{q}^{l} . We use the bulletin board $\mathcal{F}_{\text{BULLETIN}}$ to keep track of all those values that are broadcasted. Observe that no information that was posted to $\mathcal{F}_{\text{BULLETIN}}$ can ever be changed or erased.

Sharing values for the online phase. All computations during the online phase are done using additivelyshared values. 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 additively-shared among the parties, where party \mathcal{P}_i holds share α_i such that $\alpha = \sum_{i=1}^{n} \alpha_i$, and the commitments to each value are publicly known. We define the $\langle \cdot \rangle$ -representation of a shared value as follows:

Definition 2. Let $r, s, e \in \mathbb{Z}_p$, then the $\langle r \rangle$ -representation of r is defined as

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

⁸This other commitment functionality might be implemented by a hash function/random oracle, and is used whenever the linear operations of the commitment scheme are not necessary.

⁹Note that the random oracle model and $\mathcal{F}_{\text{COMMIT}}$ were already assumptions used in the original SPDZ protocol, our extra assumptions are the existence of $\mathcal{F}_{\text{BULLETIN}}$ and the DLP-hard group G.

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

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

Fig. 2: The ideal Functionality for the Bulletin board

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

Commit: On input (Commit, v, r, i, τ_v) by \mathcal{P}_i , where both v and r are either in \mathbb{Z}_p or \bot , 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 (Open, i, τ_v) by \mathcal{P}_i , the ideal functionality outputs (v, r, i, τ_v) to all players. If (NoOpen, i, τ_v) is given by the adversary, and \mathcal{P}_i is corrupt, the functionality outputs (\bot, \bot, i, τ_v) to all players.

Fig. 3: The Ideal Functionality for Commitments

where $r = \sum_{i=1}^{n} r_i$ and $\alpha \cdot r = \sum_{i=1}^{n} \gamma(r)_i$. Each player \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 $\langle r \rangle = \langle s \rangle$ if both $\langle r \rangle, \langle s \rangle$ reconstruct to the same value. Then it holds that

$$\langle r \rangle + \langle s \rangle = \langle r + s \rangle, \ e \cdot \langle r \rangle = \langle e \cdot r \rangle, \ e + \langle r \rangle = \langle e + r \rangle$$

A value that is shared as above is reconstructed or $opened^{10}$ by summing up all shares. The correctness of this opening can be checked by checking the MAC(we will use a protocol where α will not be revealed). A value $\langle a \rangle$ can either be *publicly opened* if every player \mathcal{P}_i broadcasts its share a_i , or *opened towards* \mathcal{P}_i if every other party $\mathcal{P}_j, j \neq i$ sends its share a_j to \mathcal{P}_i . Similarly, if the players *open towards* $\mathcal{F}_{\text{BULLETIN}}$ this means that they send their shares of the particular value to the bulletin board.

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 [5] and a MAC checking procedure for the output phase, this already yields an actively secure MPC scheme that is secure against up to n - 1 corrupted players¹¹.

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 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 [32]) $pc(r, r_{rand})$. When we open a $[\cdot]$ -representation, we reconstruct both r and r_{rand} .¹² This way the commitment is also opened (it is already published on $\mathcal{F}_{\text{BulleTIN}}$) and everyone can check that it is correct (but the computing parties do not need to do so).

 $^{^{10}\}mathrm{We}$ use both terms for it in this paper.

¹¹Provided that the offline phase generates valid multiplication triples and random values together with MACs.

¹²Our different flavours of opening for $\langle \cdot \rangle$ -representations can be applied here as well.

Procedure $\mathcal{P}_{PROVIDERANDOM}$

Even though we do not mention minimum lengths of seeds here, they should be chosen according to a concrete security parameter.

ProvideRandom(q, l) On input (Urandomness, q, l) from each party \mathcal{P}_i :

- (1) Each party \mathcal{P}_i commits to a seed $s_i \in \{0,1\}^*$ using $\mathcal{F}_{\text{COMMIT}}$. It also sends the commitment to $\mathcal{F}_{\text{BULLETIN}}$.
- (2) Each party opens its commitment to all parties and $\mathcal{F}_{\text{BULLETIN}}$.
- (3) Each party locally computes $s = s_1 \oplus \cdots \oplus s_n$
- (4) Each party outputs $v \leftarrow \mathcal{U}_{\boldsymbol{s}}(q, l)$.

Fig. 4: A protocol to jointly generate random values

Procedure \mathcal{P}_{Mult}

Multiply([[r]], [[s]], [[a]], [[b]], [[c]]):

- (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 $\gamma, \delta, \gamma_{rand}, \delta_{rand}$ and send these values to $\mathcal{F}_{\text{BULLETIN}}$.
- (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 [.]-shared values

Definition 3. Let $r, r_{rand} \in \mathbb{Z}_p$ and $g, h \in G$ where both g, h generate the group, then we define the $[\![r]\!]$ -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.

We define linear operations on the representations as before:

Definition 4. Let $a, b, a_{rand}, b_{rand}, e \in \mathbb{Z}_p$. We define the following linear operations on $\llbracket \cdot \rrbracket$ -sharings:

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

With a slight abuse in notation, we see that

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

$$[[r]] + [[s]] \stackrel{\circ}{=} [[r+s]], e \cdot [[r]] \stackrel{\circ}{=} [[e \cdot r]], e + [[r]] \stackrel{\circ}{=} [[e+r]]$$

In order to multiply two representations, we rely on the protocol in Figure 5 (as in [5]): Let $[\![r]\!], [\![s]\!]$ be two values where we want to calculate a representation $[\![t]\!]$ such that $t = r \cdot s$. Assume the existence of a triple $([\![a]\!], [\![b]\!], [\![c]\!])$ such that a, b are uniformly random and $c = a \cdot b$. Then one can obtain $[\![t]\!]$ using \mathcal{P}_{MULT} . Most interestingly, one does not have to perform the computations on the commitments during the online phase. Instead, only the $\langle \cdot \rangle$ -representations are manipulated!

Shared randomness from an offline phase Our online phase relies on the availability of $[\cdot]$ -representations of random values and multiplication triples. In Figure 6 we define the functionality $\mathcal{F}_{\text{SETUP}}$ that describes our preprocessing protocol, which 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. Since we assume that g, h come from a CRS, the audit is still correct in this setting.

Functionality \mathcal{F}_{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. \mathcal{A} 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 \mathcal{A} .
- (2) For all $i \in A_{BP}$, \mathcal{A} 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 $\mathcal{P}_i, i \notin A_{BP}$.
- (4) Set the flag $f = \top$.

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

Macro Bracket $(r_1, \ldots, r_n, s_1, \ldots, s_n, d)$: This macro will be run by the functionality to create $[\cdot]$ -representations. (1) Define $\mathbf{r} = \sum_{i=1}^{n} \mathbf{r}_{i}, \mathbf{s} = \sum_{i=1}^{n} \mathbf{s}_{i}.$

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

Macro Angle (r_1, \ldots, 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 $i \in A_{BP}$, \mathcal{A} inputs $\gamma_i, \mathcal{\Delta}_{\gamma} \in \mathbb{Z}_p^d$, and for $i \notin A_{BP}$, choose $\gamma_i \in_R \mathbb{Z}_p^d$ at random except for γ_i , with jbeing the smallest index not in A_{BP} (if there exists one).
- (3) If $|A_{BP}| < n \text{ set } \boldsymbol{\gamma} = \alpha \cdot \boldsymbol{r} + \boldsymbol{\Delta}_{\boldsymbol{\gamma}} \text{ and } \boldsymbol{\gamma}_{j} = \boldsymbol{\gamma} \sum_{j \neq i=1}^{n} \boldsymbol{\gamma}_{i}$, 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 output of the offline phase

The online phase The online phase of our protocol is presented in Figure 7. 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)¹³. The check of the MACs is done as in SPDZ using the protocol in Figure 8.

Security of the Online Phase 4

In this section, we will prove that for all poly-time adversaries \mathcal{A} there exists a simulator $\mathcal{S}_{\text{ONLINE}}$ such that Π_{AUDITMPC} is indistinguishable from $\mathcal{F}_{\text{AUDITMPC}}$ to every poly-time environment $\boldsymbol{\mathcal{Z}}$. As we argued before, this also implies that Π_{AUDITMPC} fulfills the *auditable correctness* requirement from Definition 1.

We start with the following Lemma from [16, Lemma 1] about correctness and soundness of the MAC check. We then prove the security of the online phase in Theorem 1.

Lemma 1. Let $p \in \mathbb{P}$. On input $(a_1, \gamma(a_1), \ldots, a_t, \gamma(a_t), p)$ $\mathcal{P}_{\text{CHECKMAC}}$ is correct and sound:

- If $\forall i : \alpha \cdot a_i = \gamma(a_i)$ then it returns 1 with probability 1.
- If $\exists i : \alpha \cdot a_i \neq \gamma(a_i)$ then it rejects except with probability 2/p.

Theorem 1. In the $\mathcal{F}_{\text{SETUP}}, \mathcal{F}_{\text{BULLETIN}}, \mathcal{F}_{\text{COMMIT}}$ -hybrid model with a random oracle, the protocol Π_{AUDITMPC} implements $\mathcal{F}_{AUDITMPC}$ with computational security against any static adversary corrupting all parties if the DLP is hard in the group G.

¹³This does not break the security, because (informally speaking) this is the same information that an \mathcal{A} receives if he corrupts n-1 parties.

Protocol $\Pi_{AUDITMPC}$

Initialize: On input (lnit, C, p) from all parties (where $p \in \mathbb{P}$ and C is a circuit over \mathbb{Z}_p , with ρ multiplication gates):

- (1) The parties send (Init, p, l) to $\mathcal{F}_{\text{Setup}}$ and obtain their shares α_i .
- (2) The parties choose the smallest $\tau \geq \rho$ such that $l|\tau$ and send (GenerateData, $m + 2, \tau$) to $\mathcal{F}_{\text{SETUP}}$. If they obtain $\rho' < \rho$ triples, they continue sending (GenerateData, 0, τ) until they obtained at least ρ triples in total.
- **Input:** On input (Input, \mathcal{I}_i , varid, x_i) by \mathcal{I}_i and (Input, \mathcal{I}_i , varid,?) from all \mathcal{P}_j , the parties and \mathcal{I}_i do the following (using a new random value [r]):
 - (1) $\llbracket r \rrbracket$ is privately opened as r, r_{rand} to \mathcal{I}_i .
 - (2) Let c_r be the commitment of [r] on $\mathcal{F}_{\text{BulleTIN}}$. \mathcal{I}_i checks that $c_r = pc(r, r_{rand})$. If not, the protocol is aborted.
 - (3) \mathcal{I}_i broadcasts $\epsilon = x_i r$ to all \mathcal{P}_j and $\mathcal{F}_{\text{BULLETIN}}$.
 - (4) All players locally compute $\llbracket x_i \rrbracket = \llbracket r \rrbracket + \epsilon$
- **Compute:** Upon input (Compute) from all \mathcal{P}_i , if **Initialize** has been executed and inputs for all input wires of C have been assigned, evaluate C gate per gate as follows:
 - Add: For two values [r], [s] with IDs $varid_r, varid_s$:
 - (1) Let $varid_t$ be a fresh ID. Each party locally computes [t] = [r] + [s] and assigns $varid_t$ to it. The commitments are excluded from the computation.
 - **Multiply:** Multiply two values [r], [s] with IDs $varid_r, varid_s$, using the multiplication triple ([a], [b], [c]).
 - (1) Let $varid_t$ be a fresh ID. The parties invoke \mathcal{P}_{MULT} . Multiply([[r]], [[s]], [[a]], [[b]], [[c]]) to compute [[t]] and assign the ID $varid_t$. The commitments are excluded from the computation.

Output: The parties open the output [y]. Let $a_1, ..., a_t$ be the values opened.

- (1) All parties compute
 - $r \leftarrow \mathcal{P}_{\text{CHECKMAC}}.CheckOutput(a_1, \gamma(a_1), \dots, a_t, \gamma(a_t), p)$ If $r \neq 0$ then stop.
- (2) All parties open the output $\llbracket y \rrbracket$ towards $\mathcal{F}_{\text{BULLETIN}}$.
- (3) All parties compute $s \leftarrow \mathcal{P}_{\text{CHECKMAC}}.CheckOutput(y, \gamma(y), y_{rand}, \gamma(y_{rand}), p)$ If $s \neq 0$ then stop. Otherwise output y.
- Audit:
 - (1) If the **Output** step was not completed, output NO AUDIT POSSIBLE.
 - (2) Run Audit for $\mathcal{F}_{\text{SETUP}}$. If it returns ACCEPT then continue, otherwise output NO AUDIT POSSIBLE.
 - (3) We follow the computation gates of the evaluated circuit C in the same order as they were computed. For the *i*-th gate, do the following:

Input: Let [r] be the opened value and $varid_x$ be the ID of input x. Set $c_{varid_x} = pc(\epsilon, 0) \cdot c$, 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 $c_{varid_t} = c_{varid_r} \cdot$ Cvarid.

- **Multiply:** The parties multiplied [r] with $varid_r$ and [s] with $varid_s$ (using the auxiliary values $[a], [b], [c], [\gamma], [\delta]$ with their respective IDs). The output has ID $varid_t$.
 - (a) Set $c_{varid_t} = c_{varid_c} \cdot c_{varid_a}^{\delta} \cdot c_{varid_b}^{\gamma} \cdot pc(\gamma \cdot \delta, 0).$
 - (b) Check that $c_{varid_r} \cdot c_{varid_a}^{-1} \stackrel{?}{=} pc(\gamma, \gamma_{rand})$ and
- $c_{varid_s} \cdot c_{varid_b}^{-1} \stackrel{?}{=} pc(\delta, \delta_{rand})$. If not, output REJECT. (4) Let y be the output of **Output** and c_y be the commitment for the output value $[\![y]\!]$.
 - If $c_y = pc(y, y_{rand})$ then output ACCEPT y.
 - If $c_y \neq pc(y, y_{rand})$ then output REJECT.

Fig. 7: The protocol for the online phase

Proof. We prove the above statement by providing a simulator S_{ONLINE} (see Figure 9). The simulator is divided for two cases, for the honest minority ($S_{\text{ONLINE, NORMAL}}$ in Figure 10) and the fully malicious setting $(\mathcal{S}_{\text{ONLINE,FULL}}$ in Figure 11).

Procedure $\mathcal{P}_{CHECKMAC}$

 $CheckOutput(v_1, \gamma(v_1), ..., v_t, \gamma(v_t), m)$ Here we check whether the MACs hold on t reconstructed values.

- (1) Each \mathcal{P}_i samples a value s_i and, to obtain the vector \mathbf{r} , invokes $\mathcal{P}_{\text{PROVIDERANDOM}}$. Provide Random(m, t) with the seed \boldsymbol{s}_i .
- (2) Each \mathcal{P}_i computes $v = \sum_{i=1}^t \mathbf{r}[i] \cdot v_i$. (3) Each \mathcal{P}_i computes $\gamma_i = \sum_{j=1}^t \mathbf{r}[j] \cdot \gamma(v_j)$ and $\sigma_i = \gamma_i \alpha_i \cdot v$.
- (4) Each \mathcal{P}_i commits to σ_i using $\mathcal{F}_{\text{COMMIT}}$ as c'_i .
- (5) Each c'_i is opened towards all players using $\mathcal{F}_{\text{COMMIT}}$.
- (6) If $\sigma = \sum_{i=1}^{n} \sigma_i$ is 0 then return 1, otherwise return 0.

Fig. 8: Procedure to check validity of MACs

Simulator S_{ONLINE}

- (1) Wait for the set A_{BP} of corrupted players from $\boldsymbol{\mathcal{Z}}$.
- (2) If $|A_{BP}| \neq n$ then forward all incoming messages that are not from $S_{\text{ONLINE,NORMAL}}$ to $S_{\text{ONLINE,NORMAL}}$, and forward all messages that come from $\mathcal{S}_{\text{ONLINE,NORMAL}}$ to the recipient.
 - If $|A_{BP}| = n$ then forward all incoming messages that are not from $S_{\text{ONLINE,FULL}}$ to $S_{\text{ONLINE,FULL}}$, and forward all messages that come from $\mathcal{S}_{\textsc{Online,full}}$ to the recipient.

Fig. 9: Simulator for the online phase

At least one honest party. The simulator runs an instance of $\Pi_{AUDITMPC}$ with the players controlled by \mathcal{Z} and simulated honest parties. For Initialize, Input, Add, Multiply it performs the same steps as in Π_{AuDITMPC} , only that it uses a fixed input 0 for the simulated honest parties during **Input**. Since every set of at most n-1 shares of a value is uniformly random and does not reveal any information about the shared secret, this cannot be distinguished from a real execution.

During **Output**, we adjust the shares of one simulated honest party to agree with the correct output y from $\mathcal{F}_{AUDITMPC}$: The simulator obtained the result y' of the simulated computation, hence it can adjust the share of a simulated honest party. Moreover, it also adjusts the MAC share as depicted in $\mathcal{S}_{\text{ONLINE,NORMAL}}$ using the MAC key α provided by $\mathcal{F}_{\text{SETUP}}$. As argued in [18], the distribution of these shares of the simulated honest parties is the same as during a protocol execution.

The commitment is information-theoretically hiding, and since the discrete logarithm $\log_q(h)$ is known to $\mathcal{S}_{\text{ONLINE,NORMAL}}$, it can compute a randomness value y_{rand} that correctly opens the commitment in $[\![y]\!]$ as posted on $\mathcal{F}_{\text{BulletIN}}$ for the value y instead of y'. It then adjusts a share of y'_{rand} such that (y', y'_{rand}) open the commitment. Once again, the distribution of this share of $y_{rand'}$ agrees with the distribution in a real protocol execution.

If moreover \mathcal{Z} decides to stop the execution, then $\mathcal{S}_{\text{ONLINE,NORMAL}}$ will forward this to the ideal functionality and $\boldsymbol{\mathcal{Z}}$ 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 from $\mathcal{F}_{AUDITMPC}$ in the simulated case. We have to show that in the real protocol, the probability that \mathcal{A} can cheat 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 [18], the first event only happens with probability 1/p. If \mathcal{A} can adjust the MACs correctly with non-negligible probability, then it can guess the secret MAC key α – which contradicts that it only holds at most n-1 shares of it which reveal no information. For the third case, Lemma 1 implies that this can only happen with probability 2/p. Since we set p to be exponential in the security parameter, the distributions are statistically indistinguishable.

Simulator $\mathcal{S}_{ ext{Online,Normal}}$
The values g, h are provided as a CRS by this simulator, so $s = \log_g(h)$ is known as well as α .
 Initialize: On input (Init, C, p) from Z: (1) Set up F_{BULLETIN} and start a local instance Π of Π_{AUDITMPC} with the dishonest parties (and simulated honest parties). (2) Run a copy of F_{SETUP}, with which Z and the simulated honest parties communicate through the simulator. (3) Run Initialize and Compute of F_{SETUP} as in Π_{AUDITMPC}. Input: On input (Input, I_i, varid, ·) by I_i and (Input, I_i, varid, ?) from and Z: If I_i is honest then follow Π_{AUDITMPC} for a fake input 0.
If \mathcal{I}_i is dishonest then extract the input value x_i from Π and send it to $\mathcal{F}_{AUDITMPC}$. Execute this step with
x_i in $\mathcal{F}_{\text{AUDITMPC}}$. Compute: Upon input (Compute) from \mathcal{Z} , if Initialize has been executed and inputs for all input gates of C have been provided, evaluate C gate per gate as follows: Add: Follow the steps of Add in Π_{AUDITMPC} . Multiply: Follow the steps of Multiply in Π_{AUDITMPC} .
Output: Obtain the output y from $\mathcal{F}_{AUDITMPC}$ and simulate $\Pi_{AUDITMPC}$ as follows:
 (1) Generate correct shares for the simulated honest parties for Π: (1.1) Let P_i be a simulated honest party and y' be the output of Π with Z right now. Let [[y']] = (⟨y'⟩, ⟨y'_{rand}⟩, c = pc(y', y'_{rand})), where y' = ∑_k y'_{o,k} and y'_{rand} = ∑_k y'_{rand,o,k}. For all honest P_j with j ≠ i, let y'_{q,j} = y'_{o,j}, y'_{rand,q,j} = y'_{rand,o,j}^a. (1.2) For P_i set y'_{q,i} = y'_{o,i} + (y - y') and γ'_{q,i} = γ'_{o,i} + α(y - y'). We have s ≠ 0, so s⁻¹ mod p exists. Set y_{rand} = (y' - y + s ⋅ y'_{rand})/s mod p, and y'_{rand,q,i} = y'_{rand,o,i} + (y_{rand} - y'_{rand}), γ'_{rand,q,i} = γ'_{rand,o,i} + α ⋅ (y_{rand} - y'_{rand}). (1.3) Let y' = ∑_k y_{a,k}, y'_{rand} = ∑_k y'_{rand} a, k, γ' = ∑_k γ'_{a,k} and γ'_{rand} = ∑_k γ'_{rand} a, k.
(2) Follow the protocol $\Pi_{AUDITMPC}$ to check the MACs according to step 1 of Output . If that step fails, let $\mathcal{F}_{AUDITMPC}$ deliver \perp to the honest parties and stop.
 (3) Send the shares of the simulated honest parties of the output [[y]] to \$\mathcal{F}_{BULLETIN}\$. If \$\mathcal{Z}\$ does not provide shares of [[y]] for all dishonest parties, then let \$\mathcal{F}_{AUDITMPC}\$ set \$y' = \perp\$ and stop. (4) Run \$\mathcal{P}_{CHECKMAC}\$ as in \$\mathcal{I}_{AUDITMPC}\$. If the MAC on the output [[y]] is correct, let \$\mathcal{F}_{AUDITMPC}\$ set \$y' = y\$,
otherwise $y' = \bot$. Audit: Run Audit as in Π_{AUDITMPC} with the malicious players. Then invoke Audit in $\mathcal{F}_{\text{AUDITMPC}}$ and output REJECT if it is the output of $\mathcal{F}_{\text{AUDITMPC}}$. If not, reveal what $\mathcal{F}_{\text{AUDITMPC}}$ outputs.
^a Similarly, the MAC keys $\gamma'_{o,j}, \gamma'_{rand,o,j}$ of those parties are not touched.

Fig. 10: Simulator for honest minority

- Audit: We focus on the two cases when $\mathcal{F}_{AUDITMPC}$ and $\Pi_{AUDITMPC}$ disagree about the output of Audit. The conditions under which $\mathcal{S}_{OFFLINE,NORMAL}$ and $\Pi_{AUDITMPC}$ output NO AUDIT POSSIBLE are the same.
 - (1) $\mathcal{F}_{\text{AUDITMPC}}$ outputs ACCEPT y when Π_{AUDITMPC} outputs REJECT does not happen due to the construction of $\mathcal{S}_{\text{OFFLINE,NORMAL}}$.
 - (2) $\mathcal{F}_{\text{AUDITMPC}}$ outputs REJECT when Π_{AUDITMPC} outputs ACCEPT y. \mathcal{A} replaced the output with \bot , but $\mathcal{P}_{\text{CHECKMAC}}$ passed successfully. This happens with probability at most 2/p according to Lemma 1.

Fully malicious setting. The intuition behind $S_{\text{ONLINE,FULL}}$ is that we let \boldsymbol{Z} send arbitrary messages during the online phase. But since all messages for $\mathcal{F}_{\text{SETUP}}$ go through $S_{\text{ONLINE,FULL}}$, we extract the used inputs after the fact which we then can use with $\mathcal{F}_{\text{AUDITMPC}}$. Observe that, since we cannot guarantee privacy, no inputs must be substituted. During the **Audit**, we run the protocol of Π_{AUDITMPC} also in the simulator (but with different outputs, as we shall see). The difference between both again is the output of **Audit** in both worlds.

- (1) $\mathcal{F}_{\text{AUDITMPC}}$ outputs ACCEPT y when Π_{AUDITMPC} outputs REJECT does not happen due to the construction of $\mathcal{S}_{\text{OFFLINE,FULL}}$.
- (2) $\mathcal{F}_{\text{AUDITMPC}}$ outputs REJECT when Π_{AUDITMPC} outputs ACCEPT y. \mathcal{Z} replaced the output with another value y' (and also y'_{rand}) that open the commitment c_y . But in step (3) of **Compute**, the simulator



already obtained y such that $pc(y, y_{rand}) = pc(y', y'_{rand})$ for some y_{rand} .¹⁴. This implies a solution of the DLP in poly-time, contradicting the assumption.

5 An Implementation of the Offline Phase

We will now provide an implementation of $\mathcal{F}_{\text{Setup}}$. It consists two phases, where we first sample correlated randomness for the online phase and then check whether the multiplication triples satisfy the required relation. We also introduce an *audit for the offline phase*. The implementation relies on a cryptosystem that allows a certain number of additions and multiplications of vectors of plaintexts (as in [18,16]). We will now recap the definition of such the scheme, where the terminology follows [18].

5.1 A Suitable Cryptosystem

We define the plaintext space \mathcal{M} as $\mathcal{M} = \mathbb{Z}_p^l$ as the direct product of $l \mathbb{Z}_p$ -instances. Let '+' and '.' be the ring operations implied by the direct product.

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 (which 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 realized. In order to map $\mathbf{m} \in \mathcal{M}$ to an element $\mathbf{a} \in \mathcal{A}$ and back, there exist the two functions

```
encode: \mathcal{M} \to \mathcal{A}decode: \mathcal{A} \to \mathcal{M}
```

¹⁴Computing y_{rand} from C and the randomization values of the inputs is straightforward. We omitted this computation here.

where *encode* is injective. We want *decode* to be the inverse of *encode* (on its image) and to be structurepreserving. Moreover, *decode* has to respect the characteristic of the field \mathbb{Z}_p and *encode* must return *short* vectors¹⁵. This is formalized as follows:

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

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

 $ParamGen(1^{\lambda}, \mathcal{M})$ The algorithm outputs the dimension N of the ring \mathcal{A} and descriptions for *encode* and *decode* as well as a randomized algorithm D_{ρ}^{d} , an additive abelian group \mathcal{B} and a set of allowable circuits $C. D_{\rho}^{d}$ outputs vectors $\mathbf{r} \in \mathbb{Z}^{d}$ such that $Pr[||\mathbf{r}||_{\infty} \geq \rho \mid \mathbf{r} \leftarrow D_{\rho}^{d}] < negl(\lambda)$. \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}_p^l , 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 $\hat{f} \in C$ evaluates the function $f \ l$ times on inputs in $(\mathbb{Z}_p)^{n(f)}$ in parallel.

 $Enc_{pk}(\boldsymbol{x},\boldsymbol{r})$ 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 $Enc_{pk}(\boldsymbol{m}) = Enc_{pk}(\boldsymbol{x}',\boldsymbol{r}')$ with $\boldsymbol{x}' \leftarrow encode(\boldsymbol{m})$ and $\boldsymbol{r}' \leftarrow D_{\rho}^d$.

For the ZKPoPKs, we require that Enc_{pk} is homomorphic for at least a *small* number V of correct ciphertexts. More formally: Let $\boldsymbol{x}_1, ..., \boldsymbol{x}_V \in image(encode), \ \boldsymbol{r}_1, ..., \boldsymbol{r}_V \leftarrow D_{\rho}^d$. Then it holds that

 $Enc_{pk}(\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_V, \boldsymbol{r}_1 + \ldots + \boldsymbol{r}_V) = Enc_{pk}(\boldsymbol{x}_1, \boldsymbol{r}_1) \oplus \ldots \oplus Enc_{pk}(\boldsymbol{x}_V, \boldsymbol{r}_V)$

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.

 $Dec_{sk}(g)$: For $g \in \mathcal{B}$ this algorithm will return an $m \in \mathcal{M} \cup \{\bot\}$.

KeyGen(): This algorithm samples a public key/private key pair (pk, sk).

- KeyGen^{*}: A meaningless public key \overline{pk} is returned. Let $(pk, \cdot) \leftarrow KeyGen()$ and $m \in \mathcal{M}$ be arbitrary. Then \overline{pk} will be such that
 - (1) $Enc_{\overline{nk}}(\boldsymbol{m})$ and $Enc_{\overline{nk}}(\boldsymbol{0})$ are statistically indistinguishable.
 - (2) pk and \overline{pk} are computationally indistinguishable.

Correctness. Let $n(f), f \in C$ be the number of input values of f and let \hat{f} be the embedding of f into \mathcal{B} where '+' is replaced by \oplus , '.' by \otimes and the constant $c \in \mathbb{Z}_p$ by $Enc_{pk}(encode(c), \mathbf{0})$. 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.

¹⁵given a suitable basis for \mathcal{A} that allows us define a norm $|| \cdot ||_{\infty}$.

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

Key generation:

- (1) When receiving (StartKeyGen) from all parties, run $P \leftarrow ParamGen(1^{\lambda}, \mathcal{M})$.
- (2) Wait for randomness r_i from every party \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 \mathcal{P}_i .

Distributed decryption:

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

Fig. 12: The ideal functionality for distributed key generation and decryption

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

$$\begin{aligned} \Pr\left[Dec_{sk}(\boldsymbol{c}) \neq f\left(decode(\boldsymbol{x}_{1}), ..., decode(\boldsymbol{x}_{n(f)})\right) \middle| & P \leftarrow ParamGen(1^{\lambda}, \mathcal{M}) \land (pk, sk) \leftarrow KeyGen() \land \\ & f \in C \land (\boldsymbol{x}_{1}, ..., \boldsymbol{x}_{n(f)}, \boldsymbol{r}_{1}, ..., \boldsymbol{r}_{n(f)}) \in \mathcal{A}^{n(f)} \times \mathbb{Z}^{n(f)} \land \\ & \boldsymbol{c} \leftarrow \widehat{f}(\boldsymbol{c}_{1}, ..., \boldsymbol{c}_{n(f)}) \land \left(decode(\boldsymbol{x}_{i}) \in \mathcal{M} \land \\ ||\boldsymbol{x}_{i}||_{\infty} \leq B_{plain} \land ||\boldsymbol{r}_{i}||_{\infty} \leq B_{rand} \land \\ & \boldsymbol{c}_{i} \leftarrow Enc_{pk}(\boldsymbol{x}_{i}, \boldsymbol{r}_{i})\right)_{i \in [n(f)]} \right] < negl(\lambda) \end{aligned}$$

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

Distributed decryption and key generation. We require that the cryptosystem supports distributed key generation and decryption, as captured in $\mathcal{F}_{\text{KeyGenDec}}$.

Definition 5 (Admissible Cryptosystem). Let C contain formulas of the form

$$\left(\sum_{i=1}^{n} x_{i}\right) \cdot \left(\sum_{i=1}^{n} y_{i}\right) + \sum_{i=1}^{n} z_{i}$$

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 $\mathcal{F}_{KeyGenDec}$.

The parameter $sec \in \mathbb{N}$ in the above definition comes from the ZKPoPKs that we will see later. One can easily see that e.g. the Ring-LWE-based BGV scheme [10] or the BGH extension of LWE-based BGV [9] have the required features. For more details, we once again refer to [18,16]

5.2 Zero-Knowledge Proofs of Plaintext Knowledge

If a shared value is reconstructed during the online phase of the protocol, then the related commitment should be opened to the same value. We ensure this (and the correct generation of ciphertexts) during the offline phase using NIZKs. Given the security parameter *sec*, $2 \cdot sec$ ciphertexts $c_1, ..., c_{2 \cdot sec} \in image(Enc_{pk}(\cdot))$ and $sec \cdot l$ group elements $d_{1,1}, ..., d_{sec,l} \in G$, we prove the following relation:

$$R_{CTC} = \left\{ (\boldsymbol{a}, \boldsymbol{w}) \mid \boldsymbol{a} = (\boldsymbol{c}_{1}, ..., \boldsymbol{c}_{2 \cdot sec} \land d_{1,1}, d_{2,1}, ..., d_{sec,l}, pk) \land \boldsymbol{w} = (\boldsymbol{x}_{1}, \boldsymbol{r}_{1}, ..., \boldsymbol{x}_{2 \cdot sec}, \boldsymbol{r}_{2 \cdot sec}) \land \right. \\ \left[\boldsymbol{c}_{i} = Enc_{pk}(\boldsymbol{x}_{i}, \boldsymbol{r}_{i}) \land \boldsymbol{c}_{sec+i} = Enc_{pk}(\boldsymbol{x}_{sec+i}, \boldsymbol{r}_{sec+i}) \land \right. \\ \left. ||\boldsymbol{x}_{i}||_{\infty} \leq B_{plain} \land ||\boldsymbol{x}_{sec+i}||_{\infty} \leq B_{plain} \land decode(\boldsymbol{x}_{i}) \in \mathbb{Z}_{p}^{l} \land \\ \left. decode(\boldsymbol{x}_{sec+i}) \in \mathbb{Z}_{p}^{l} \land ||\boldsymbol{r}_{i}||_{\infty} \leq B_{rand} \land ||\boldsymbol{r}_{sec+i}||_{\infty} \leq B_{rand} \land \\ \left. \left(d_{i,j} = pc(decode(\boldsymbol{x}_{i})[j], decode(\boldsymbol{x}_{sec+i})[j]) \right)_{j \in [l]} \right]_{i \in [sec]} \right\} \right\}$$

To prove this statement, we execute two instances of Π_{ZKPOPK} from [18] for $c_1, ..., c_{sec}$ and

 $c_{sec+1}, ..., c_{2 \cdot sec}$ simultaneously with the same randomness. The *blinding* values of the proof will also be put into commitments (as well as for the randomness) to prove both that we can open the commitments and that their opening values are equal to the plaintexts of the encryptions. We use an optimization for the proofs due to [28] called *proof with abort*, which yields smaller parameters for the cryptosystem using the zero-knowledge proofs with the Fiat Shamir heuristic ([20]). Moreover, its necessary to use this heuristic to get good randomness into the proof - since both the sender and the receiver might be corrupted in the fully malicious setting.

For the proof, we use the same notation as SPDZ: Let $\mathbf{R} \in \mathbb{Z}^{sec \times d}$ be the matrix whose *i*th row is \mathbf{r}_i of \mathbf{c}_i and \mathbf{R}' the similar matrix for \mathbf{r}_{sec+i} . Moreover, let $V = 2 \cdot sec - 1$. For a vector $\mathbf{e} \in \{0, 1\}^{sec}$ we define the matrix $\mathbf{M}_{\mathbf{e}} \in \mathbb{Z}_2^{V \times sec}$ as

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

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 $\boldsymbol{\overline{d}} = (d_{1,1}, d_{2,1}, ..., d_{sec,l})$.

Given the protocol Π_{ZKPOPK} from [18] (which is a honest-verifier zero-knowledge proof of knowledge for a part of our relation) the following statement is straightforward:

Remark 3. The protocol Π_{CTC} is an honest-verifier zero- knowledge proof of knowledge for the relation R_{CTC} .

Proof. We observe that the protocol Π_{CTC} runs three instances of the SPDZ proof Π_{ZKPoPK} in parallel, two for the ciphertext vectors c, c' and one for the commitments \overline{d} . Correctness, soundness and honest-verifier zero- knowledge follow directly for the first two instances due to the proof in [18], as we use different blinding values a, 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) Consider the connection between the plaintext values and the commitments. First of all, we use an instance of the BeDOZa proof [7], 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 follows from the fact that our initially chosen blinding is the same for the ciphertexts \mathbf{y}, \mathbf{y}' and the commitments \mathbf{q} . We also observe that the same randomness M_e is used for both cases, hence the operations on the ciphertexts carry over directly to the commitments.

The protocol $\Pi_{\rm CTC}$

- (1) For $i \in \{1, ..., V\}$ the prover generates $y_i, y'_i \in \mathbb{Z}^l$ and $s_i, s'_i \in \mathbb{Z}^d$ as follows: Let s_i, s'_i be random such that $||\mathbf{s}_i||_{\infty}, ||\mathbf{s}'_i||_{\infty} \leq 128 \cdot d \cdot \rho \cdot sec^2$. For $\mathbf{y}_i, \mathbf{y}'_i$, let $\mathbf{m}_i, \mathbf{m}'_i \in \mathbb{Z}^l_p$ be random elements and set $\mathbf{y}_i = encode(\mathbf{m}_i) + \mathbf{u}_i$, $y'_i = encode(m'_i) + u'_i$ where both u_i, 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 $\boldsymbol{a}_i \leftarrow Enc_{pk}(\boldsymbol{y}_i, \boldsymbol{s}_i), \boldsymbol{a}'_i \leftarrow Enc_{pk}(\boldsymbol{y}'_i, \boldsymbol{s}'_i)$ and $q_{i,j} \leftarrow pc(decode(\boldsymbol{m}_i)[j], decode(\boldsymbol{m}'_i)[j])$ 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 *i*th column is s_i and S' to have s'_i as *i*th column respectively. Moreover, let $y \leftarrow (y_1, ..., y_V)$, $y' \leftarrow (y'_1, ..., y'_V), a \leftarrow (a_1, ..., a_V)$ and $a' \leftarrow (a'_1, ..., a'_V)$. For the commitments, we define $q_i \leftarrow (q_{i,1}, ..., q_{1,l})$ 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 $e \leftarrow \mathcal{U}(2, sec)$ from the random oracle with seed $a||a'||c||c'||\overline{d}||q$.
- (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 $128 \cdot \tau \cdot N \cdot sec^2 - \tau \cdot sec$ or the ∞ -norm of any value of T, 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 (z, z', T, T') to the verifier.
- (6) The verifier obtains e from $\mathcal{U}(2, sec)$ using the seed $a||a'||c||c'||\overline{d}||q$. Let t_i be the *i*th row of T and t'_i the respective row of T'. He computes $f_i \leftarrow Enc_{pk}(z_i, t_i), f'_i \leftarrow Enc_{pk}(z'_i, t'_i)$ and sets
- $f \leftarrow (f_1, ..., f_V), f' \leftarrow (f'_1, ..., f_V)$. In addition, the verifier computes the commitments $g_{i,j} = pc(decode(\boldsymbol{z}_i)[j], decode(\boldsymbol{z}_i')[j]) \text{ for } i \in \{1, ..., V\}, j \in \{1, ..., l\}.$
- (7) The verifier checks whether $decode(\boldsymbol{z}_i) \in \mathbb{Z}_p^l$, $decode(\boldsymbol{z}'_i) \in \mathbb{Z}_p^l$ and whether all of the following conditions hold: (7.1) $\boldsymbol{f}^{\top} = \boldsymbol{a}^{\top} \oplus (\boldsymbol{M}_{\boldsymbol{e}}\boldsymbol{c}^{\top})$

$$(7.2) \quad \boldsymbol{f'}^{\top} = \boldsymbol{a'}^{\top} \oplus (\boldsymbol{M_e c'}^{\top})$$

- (7.2) $\boldsymbol{f'} = \boldsymbol{a'} \oplus (\boldsymbol{M_ec'})$ (7.3) $||\boldsymbol{z}_i||_{\infty}, ||\boldsymbol{z}'_i||_{\infty} \le 128 \cdot N \cdot \tau \cdot \sec^2$
- (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 *i*th row of \boldsymbol{M}_e . Check that $\forall i \in \{1, ..., sec\} \; \forall j \in \{1, ..., l\} : g_{i,j} = q_{i,j} \cdot \prod_{k=1}^{sec} (d_{k,i}^{m_i[k]})$
- (7.6) If all these conditions hold, then the verifier *accepts*. Otherwise he *rejects*.

Fig. 13: The protocol for the zero-knowledge proof of plaintext knowledge

- (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 a, a', 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 a, a', e, z, z', T, 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 q and thereby the whole transcript. Due to the linearity (and because there is only one commitment for a combination of shared value & randomness) the commitments are also as in the protocol.

Observe that $\Pi_{\rm CTC}$ 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.

Resharing Plaintexts Among Parties 5.3

In order to compute the secred shared MAC, we compute the product of the shared value and the secret MAC key α using the homomorphic encryption scheme and share the result. To perform this resharing, the Procedure $\mathcal{P}_{\text{Reshare}}$

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

- (1) Each \mathcal{P}_i samples a uniformly random $f_i \in \mathbb{Z}_p^l$. We denote $f := \sum_{j=1}^n f_j$
- (2) Each \mathcal{P}_i computes and broadcasts $e_{f_i} \leftarrow Enc_{pk}(f_i)$ to all parties and $\mathcal{F}_{\text{BulleTIN}}$.
- (3) Each \mathcal{P}_i proves with a ZKPoPK that e_{f_i} is (B_{plain}, B_{rand}, C) -admissible using the Random Oracle version of Π_{ZKPOPK} . It sends the proof to $\mathcal{F}_{\text{BULLETIN}}$.
- (4) The players compute $e_f = \bigoplus_{i=1}^n e_{f_i}$, set $e_{m+f} = e_m \oplus e_f$ and check the ZKPoPKs. If they are not correct, then they abort.
- (5) The players decrypt e_{m+f} to obtain m + f publicly.
- (6) \mathcal{P}_1 sets $m_1 = m + f f_1$ and each other player \mathcal{P}_i sets $m_i = -f_i$.

Fig. 14: A procedure that shares the plaintext of a publicly encrypted value

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

 $\mathcal{P}_{\text{COMRESHARE}}(e_{m}, e_{r,1}, ..., e_{r,n}, r_{1}, ..., r_{n}):$

- (1) Each \mathcal{P}_i samples a uniformly random $f_i \in \mathbb{Z}_p^l$. We denote $f := \sum_{j=1}^n f_j$
- (2) Each \mathcal{P}_i computes and broadcasts $e_{f_i} \leftarrow Enc_{pk}(f_i)$ to all parties and $\mathcal{F}_{\text{BulleTIN}}$.
- (3) For each $k \in \{1, \ldots, l\}$, each party \mathcal{P}_i publishes $c_{f,i,k} \leftarrow pc(f_i[k], -r_i[k])$ on $\mathcal{F}_{\text{BULLETIN}}$.
- (4) Each \mathcal{P}_i proves with a ZKPoPK using Π_{CTC} that $e_{f_i}, e_{r,i}$ are (B_{plain}, B_{rand}, C) -admissible and that the commitments hold. It sends the proof transcript to $\mathcal{F}_{\text{BULLETIN}}$.
- (5) Each player checks whether the proofs are valid.
- (6) The players locally compute $e_f = \bigoplus_{i=1}^n e_{f_i}$ and set $e_{m+f} = e_m \oplus e_f$.
- (7) The players decrypt e_{m+f} using $\mathcal{F}_{\text{KeyGenDec}}$ to obtain m+f.
- (8) \mathcal{P}_1 sets $m_1 = m + f f_1$ and each other player \mathcal{P}_i sets $m_i = -f_i$.
- (9) For $k \in \{1, ..., l\}$, \mathcal{P}_1 sets $c'_{m,1,k} = pc((m+f)[k], 0)/c_{f,1,k}$ and all other players \mathcal{P}_i set $c'_{m,i,k} = c^{-1}_{f,i,k}$.
- (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. 15: A procedure that shares the plaintext of a publicly encrypted value together with a commitment

original SPDZ protocol uses the procedure $\mathcal{P}_{\text{ResHARE}}$, which is depicted in Figure 14. The 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 $\mathcal{P}_{\text{ResHARE}}$ are straightforward and can be verified in [18]:

Remark 4. Assuming a (B_{plain}, B_{rand}, C) -admissible cryptosystem and $\mathcal{F}_{BULLETIN}$, then the following statements are true about $\mathcal{P}_{RESHARE}$ in the Random Oracle model:

- (1) If all parties honestly follow the protocol, then they obtain correct and randomly distributed shares of the plaintext of e_m w.h.p.
- (2) If at least one and at most all parties are corrupted and the ZKPoPKs are correct, then the obtained shares might not be correct (with respect to *m*), but the parties can obtain a ciphertext that contains the shared value w.h.p.

The procedure in $\mathcal{P}_{\text{ResHARE}}$ is sufficient as long as one does not have to generate commitments. To add them, we introduce the new procedure $\mathcal{P}_{\text{COMRESHARE}}$ which can be found in Figure 15. We now give a similar characterization about $\mathcal{P}_{\text{COMRESHARE}}$ like in Remark 4:

Remark 5. Assuming a (B_{plain}, B_{rand}, C) -admissible cryptosystem, $\mathcal{F}_{\text{BULLETIN}}$ and a group G where the DLP is hard, then the following statements are true about $\mathcal{P}_{\text{COMRESHARE}}$ in the Random Oracle model:

(1) If all parties honestly follow the protocol then they obtain correct and randomly distributed shares of the plaintext of e_m and correct commitments for their shares (with randomness from the $e_{r,i}$) w.h.p.

Procedure $\mathcal{P}_{DATACHECK}$

 $CheckTriples(t_1, ..., t_{2\kappa})$: We put the triples into the checking and evaluation vectors C and O. Then, correctness is established using the same trick as in $\mathcal{P}_{CHECKMAC}$. For a vector of triples C, we want to access all *i*th $\llbracket \cdot \rrbracket$ -representations in vector form as C(i).

- (1) Let $\boldsymbol{C} \leftarrow (t_1, ..., t_{\kappa})$ and $\boldsymbol{O} \leftarrow (t_{\kappa+1}, ..., t_{2\kappa})^{a}$.
- (2) The parties execute $\mathcal{P}_{\text{ProvideRandom}}$. Provide Random (p, κ) using seeds s_i to generate the joint random vector t.
- (3) Calculate $\gamma = \mathbf{t} \odot \mathbf{O}(1) \mathbf{C}(1)$ and $\boldsymbol{\Delta} = \mathbf{O}(2) \mathbf{C}(2)$ locally.
- (4) Open γ and Δ 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 v.
- (7) Each party locally reconstructs v.
- (8) For all positions i of v that are 0, output 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. 16: A procedure to check the validity of triples

- (2) If at least one and at most all parties are corrupted and the ZKPoPKs are correct, then the obtained shares might not yield a correct secret sharing of m, but the parties know how to open the ciphertexts and commitments w.h.p.
- (3) If at least one and at most all parties are corrupted and the ZKPoPKs are correct, then m and m' might be different. The parties know a sharing of m', and $e_{m'}$ 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 4 and Theorem 3.

5.4 Generating and Checking Triples

Our protocol does not have a reliable decryption algorithm for encrypted values. This means that \mathcal{A} will always be able to influence the outcome of the decryption process. But we use the homomorphic property of the encryption scheme to generate triples of random values, which we store as secret shared values.

To check the correctness of the triples during the offline phase, we employ a similar technique as in [16] and sacrifice a triple to check the correctness of another one. This is formally done in procedure $\mathcal{P}_{DATACHECK}$ as depicted in Figure 16.

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 Random Oracle model, the test CheckTriples is correct and an adversary corrupting all parties can pass the test CheckTriples for κ triples, out of which at least one is not correct, with probability at most $\kappa/|\mathbb{Z}_p|$.

Proof. Correctness can 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 $1/|\mathbb{Z}_p|$.

Procedure $\mathcal{P}_{DATAGEN}$

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_a an encryption of $a \in \mathcal{M}$. Note that e_{α} 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 $h = \lfloor T/l \rfloor$, then for each $j \in \{1, \ldots, h\}$ the parties do the following:

- (1) Each party \mathcal{P}_i samples uniformly random $r_i, s_i \in \mathcal{M}$, calculates $e_{r,i} \leftarrow Enc_{pk}(r_i), e_{s,i} \leftarrow Enc_{pk}(s_i)$ and broadcasts $e_{r,i}, e_{s,i}$ to all players and $\mathcal{F}_{\text{BULLETIN}}$.
- (2) For each $k \in \{1, \ldots, l\}$, each party \mathcal{P}_i publishes $c_{r,i,k} \leftarrow pc(\mathbf{r}_i[k], \mathbf{s}_i[k])$ on $\mathcal{F}_{\text{BULLETIN}}$.
- (3) Each party \mathcal{P}_i invokes Π_{CTC} on $e_{r,i}, e_{s,i}, \{c_{r,i,k}\}_{k \in \{1,\dots,l\}}$ 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 $e_r = \bigoplus_i e_{r,i}, e_s = \bigoplus_i e_{s,i}$ 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(κ, l): The same as for RandomValues, but the parties additionally multiply values to generate triples. Set $h = \lceil \kappa/l \rceil$, then for $j \in \{1, \ldots, h\}$ the parties do the following:

- (1) Each party \mathcal{P}_i samples uniformly random $\mathbf{a}_i, \mathbf{b}_i, \mathbf{f}_i, \mathbf{g}_i, \mathbf{h}_i \in \mathcal{M}$, calculates $e_{\mathbf{a},i} \leftarrow Enc_{pk}(\mathbf{a}_i)$, $e_{\mathbf{b},i} \leftarrow Enc_{pk}(\mathbf{b}_i)$ as well as $e_{\mathbf{f},i} \leftarrow Enc_{pk}(\mathbf{f}_i), e_{\mathbf{g},i} \leftarrow Enc_{pk}(\mathbf{g}_i), e_{\mathbf{h},i} \leftarrow Enc_{pk}(\mathbf{h}_i)$ and broadcasts the ciphertexts to all players and $\mathcal{F}_{\text{BULLETIN}}$.
- (2) For each $k \in \{1, \ldots, l\}$, each party \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 \mathcal{P}_i provides a ZKPoPK for $(\boldsymbol{a}_i, \boldsymbol{f}_i, (c_{a,i,k})_{k \in \{1,...,l\}})$ and $(\boldsymbol{b}_i, \boldsymbol{g}_i, (c_{b,i,k})_{k \in \{1,...,l\}})$ using Π_{CTC} and sends the transcript to $\mathcal{F}_{\text{BulletIN}}$.
- (4) Each party \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 $e_{\mathbf{a}} = \bigoplus_{i} e_{\mathbf{a},i}$ and $e_{\mathbf{b}} = \bigoplus_{i} e_{\mathbf{b},i}$.
- (6) The parties compute $e_{\boldsymbol{a}\cdot\boldsymbol{b}} = e_{\boldsymbol{a}} \otimes e_{\boldsymbol{b}}$ and invoke $\mathcal{P}_{\text{COMRESHARE}}(e_{\boldsymbol{a}\cdot\boldsymbol{b}}, (e_{\boldsymbol{h},i})_{i \in \{1,...,n\}}, (\boldsymbol{h}_i)_{i \in \{1,...,n\}})$. As a result, each party \mathcal{P}_i obtains shares c_i and all parties obtain a ciphertext e_c such that $c = \sum_i c_i$.
- (7) Locally compute $e_{\mathbf{f}} = \bigoplus_{i} e_{\mathbf{f},i}, e_{\mathbf{g}} = \bigoplus_{i} e_{\mathbf{g},i}$ and $e_{\mathbf{h}} = \bigoplus_{i} e_{\mathbf{h},i}$. The parties compute the product of e_{α} with $e_{\alpha}, e_{b}, e_{c}, e_{f}, e_{g}, e_{h}$ and invoke $\mathcal{P}_{\text{RESHARE}}(\cdot)$ on each such product to distribute a sharing of the MAC on each such value.

Fig. 17: Procedure $\mathcal{P}_{DATAGEN}$ to generate both triples and random values

If \mathcal{A} 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 $\kappa/|\mathbb{Z}_p|$ for *CheckTriples*.

In addition, we need a procedure that actually generates random values and triples in our desired $[\cdot]$ -representation. Figure 17 specifies this procedure $\mathcal{P}_{DATAGEN}$ formally.

5.5 The Offline Phase

In this subsection, we present the protocol Π_{Setup} which describes our full offline phase. We decided to exclude most of the well-known protocol steps from [18,16] to focus on the *audit* part of Π_{Setup} . Basically, the audit follows the computation and ensures that

- (1) All encrypted values and commitments have zero-knowledge proofs.
- (2) All zero-knowledge proofs are valid.
- (3) Linear operations on commitments are carried out correctly.

Protocol Π_{Setup}

This procedure sets up the cryptosystem for the protocol. Moreover, the random data for Π_{AUDITMPC} is generated that is needed during execution.

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

- (1) The parties use $\mathcal{F}_{\text{KeyGenDec}}$ 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 \mathcal{P}_i generates a private $\alpha_i \in \mathbb{Z}_p$. Let $\alpha = \sum_{j=1}^n \alpha_j$.
- (4) Each \mathcal{P}_i computes and broadcasts $e_{\alpha_i} = Enc_{pk}(diag(\alpha_i))$.
- (5) Each player \mathcal{P}_i uses Π_{ZKPoPK} to prove that e_{α_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 $e_{\alpha} = \bigoplus_{i=1}^{n} e_{\alpha_i}$.

Compute: On input (GenerateData, T, κ) from all parties and if l divides T and κ , the players execute the subprocedures of $\mathcal{P}_{\text{DATAGEN}}$. Afterwards they check the results for correctness using $\mathcal{P}_{\text{DATACHECK}}$.

- (1) $(\llbracket r_1 \rrbracket, ..., \llbracket r_T \rrbracket) \leftarrow \mathcal{P}_{\text{DATAGEN}}.RandomValues(T, l)$
- (2) $(t_1, ..., t_{\kappa}) \leftarrow \mathcal{P}_{\text{DATAGEN}}.Triples(\kappa, l)$
- (3) $(v_1, ..., v_{\kappa'}) \leftarrow \mathcal{P}_{\text{DATACHECK}}.CheckTriples(t_1, ..., t_{\kappa})$
- (4) Return $(\llbracket r_1 \rrbracket, ..., \llbracket r_T \rrbracket, v_1, ..., v_{\kappa'}).$

Audit: If Compute was executed successfully, do the following together with $\mathcal{F}_{\text{BULLETIN}}$:

- (1) Obtain all *ids* and messages on $\mathcal{F}_{\text{Bulletin}}$.
- (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 REJECT.
- (3) For every transcript of Π_{CTC} or Π_{ZKPOPK} , check whether the values for each instance are on $\mathcal{F}_{\text{BulletIN}}$. Otherwise return REJECT.
- (4) Run the verifier part for each transcript of Π_{CTC} , Π_{ZKPOPK} . If the verifier rejects, return REJECT.
- (5) For each value [a] that was generated with $\mathcal{P}_{\text{DATAGEN}}$, check whether its commitment can be obtained from the commitments to the shares as in $\mathcal{P}_{\text{DATAGEN}}$. If not, return REJECT.
- (6) Run $\mathcal{P}_{\text{DATACHECK}}$ on the commitments of the triples simulating the invocation of $\mathcal{P}_{\text{PROVIDERANDOM}}$. If one of the triples that were returned by **Compute** does not open to 0 in the sanity check, return REJECT.
- (7) Check for every opened value r with randomness s and commitment c whether c = pc(r, s). If not, return REJECT.
- (8) Return ACCEPT.

Fig. 18: Protocol Π_{Setup} that performs the preprocessing for the online phase

- (4) The procedure $\mathcal{P}_{\text{DATACHECK}}$ was executed correctly.
- (5) All opened commitments are indeed correctly opened.

To ease notation, we define the function diag as $diag : \mathbb{Z}_p \to \mathcal{M}, a \mapsto (a, a, ..., a)$. The offline phase is

described in Figure 18.

6 Security Proof of the Offline Phase

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

Theorem 2. 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 ($\mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{BULLETIN}}$)-hybrid model with a Random Oracle if the DLP is hard in the group G.

Proof. Consider the simulator in Figure 19, we will now prove that Π_{Setup} is computationally indistinguishable from $S_{\text{OFFLINE}} \diamond \mathcal{F}_{\text{Setup}}$. 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

Simulator $\mathcal{S}_{OFFLINE}$

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 $S_{\text{OFFLINE,NORMAL}}$ to $S_{\text{OFFLINE,NORMAL}}$, and send all messages that come from $S_{\text{OFFLINE,NORMAL}}$ to the proper recipient.

If $|A_{BP}| = n$, then forward all incoming messages that are not from $S_{\text{OFFLINE,FULL}}$ to $S_{\text{OFFLINE,FULL}}$, and send all messages that come from $S_{\text{OFFLINE,FULL}}$ to the proper recipient.

Fig. 19: Simulator for the offline phase

Simulator $S_{OFFLINE,FULL}$, 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 its own random oracle \mathcal{U} 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}_{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 $e_{\alpha,i}$ from all parties \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 $e_{\alpha,i}$ to obtain α_i .
- (6) Send the α_i to $\mathcal{F}_{\text{SETUP}}$ and compute $\alpha = \sum_i \alpha_i$. Moreover, compute locally $e_{\alpha} = \bigoplus_i e_{\alpha,i}$.

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

Fig. 20: Partial simulator for the offline phase, fully malicious

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.

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 $S_{OFFLINE,FULL}$ 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 $\boldsymbol{\mathcal{Z}}$ to choose a 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.

Simulator $S_{\text{OFFLINE,FULL}}$ Part 2

Compute:

- (1) The simulator waits for the ciphertexts $e_{r,i}, e_{s,i}$, 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 $r_i \leftarrow Dec_{sk}(e_{r,i}), s_i \leftarrow Dec_{sk}(e_{s,i})$ and sends them to $\mathcal{F}_{\text{SETUP}}$.
- (3) Compute $\Delta_r[k] = pc(\sum_i r_i[k], \sum_i s_i[k])/c_{r,k}$ for all $k \in \{1, ..., l\}$. If the ZKPoPKS are not all correct, then let Δ_r be random values from G.
- (4) Locally compute $e_{\mathbf{r}} = \bigoplus_{i} e_{\mathbf{r},i}, e_{\mathbf{s}} = \bigoplus_{i} e_{\mathbf{s},i}$ as well as $e_{\alpha \mathbf{r}} = e_{\alpha} \otimes e_{\mathbf{r}}$ and $e_{\alpha \mathbf{s}} = e_{\alpha} \otimes e_{\mathbf{s}}$.
- (5) Do the following for $x = \alpha r$ and then $x = \alpha s$:
 - (5.1) Wait for $e_{f,i}$ from each party \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 $\boldsymbol{e}_{\boldsymbol{a}}$ using $\mathcal{F}_{\text{KeyGenDec}}$. Then let $\boldsymbol{\gamma}_1 \leftarrow \boldsymbol{a}' Dec_{sk}(\boldsymbol{e}_{\boldsymbol{f},1})$ and $\boldsymbol{\gamma}_i \leftarrow -Dec_{sk}(\boldsymbol{e}_{\boldsymbol{f},i})$ 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 $e_{a,i}, e_{b,i}, e_{f,i}, e_{g,i}, e_{h,i}$, commitments $c_{a,i,k}, c_{b,i,k}$ and ZKPoPKs from all parties \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 $\mathbf{a}_i \leftarrow Dec_{sk}(e_{\mathbf{a},i}), \mathbf{b}_i \leftarrow Dec_{sk}(e_{\mathbf{b},i}), \mathbf{f}_i \leftarrow Dec_{sk}(e_{\mathbf{f},i}), \mathbf{g}_i \leftarrow Dec_{sk}(e_{\mathbf{f},i}), \mathbf{a}_i \leftarrow Dec_{sk}(e_{\mathbf{f},i})$.
- (9) Locally compute $e_{\mathbf{a}} = \bigoplus_i e_{\mathbf{a},i}, e_{\mathbf{b}} = \bigoplus_i e_{\mathbf{b},i}, e_{\mathbf{h}} = \bigoplus_i e_{\mathbf{h},i}$ and compute $e_{\mathbf{a}\cdot\mathbf{b}} = e_{\mathbf{a}} \otimes e_{\mathbf{b}}$.
- (10) Wait for $e'_{c,i}$ from each party \mathcal{P}_i , the commitments $(c_{c',i,k})_{k \in \{1,...,l\}}$ (using $-\mathbf{h}_i$ 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 $e'_{c} = \bigoplus_{i} e'_{c,i}$ and $e_{a \cdot b + c} = e'_{c} \oplus e_{a \cdot b}$
- (12) Wait for the decryption \mathbf{a}' of $e_{\mathbf{a}\cdot\mathbf{b}+\mathbf{c}}$ using $\mathcal{F}_{\text{KeyGenDec}}$.
- (13) Let $c_1 \leftarrow a' Dec_{sk}(e'_{c,1})$ and $c_i \leftarrow -Dec_{sk}(e'_{c,i})$ for $i \in \{2, ..., n\}$. Moreover, set $e_c = Enc_{pk}(a') \bigoplus_i e'_{c,i}$ with some standard randomness.
- (14) For $k \in \{1, ..., l\}$ compute $c_{c,1,k} = pc(a'[k], 0)/c_{c',1,k}$ and $c_{c,i,k} = c_{c',i,k}^{-1}$ 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 $\boldsymbol{\Delta}_{a}[k] = pc(\sum_{i} \boldsymbol{a}_{i}[k], \sum_{i} \boldsymbol{f}_{i}[k])/c_{a,k}, \ \boldsymbol{\Delta}_{b}[k] = pc(\sum_{i} \boldsymbol{b}_{i}[k], \sum_{i} \boldsymbol{g}_{i}[k])/c_{b,k} \text{ and } \boldsymbol{\Delta}_{c}[k] = pc(\sum_{i} \boldsymbol{c}_{i}[k], \sum_{i} \boldsymbol{h}_{i}[k])/c_{c,k} \text{ for all } k \in \{1, \dots, l\}.$
- (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 $\mathcal{P}_{DATACHECK}$ to \mathcal{F}_{SETUP} .

^aThis is unrelated to this particular proof, but it will make **Audit** fail as we want. ^bI.e. it provides randomness using its version of \mathcal{U} .

Fig. 21: 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 its random oracle \sqcap 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 $e_{\alpha,i}$ for all $\mathcal{P}_i, i \in A_{BP}$ and the respective ZKPoPKs using Π_{ZKPoPK} . If the proofs are not correct, stop the execution of **Initialize** here. Otherwise, decrypt all $e_{\alpha,i}$ 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}_{SETUP} with (Audit). Return the value of \mathcal{F}_{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 $e_{r,i}, e_{s,i}$ to obtain the vectors r_i, s_i .
 - in step 6 it calls *SReshare* for both e_r, e_s to obtain $\Delta_{\gamma,r}, \Delta_{\gamma,s}, \gamma_{r,i}, \gamma_{s,i}$
 - call RandomValues on $\mathcal{F}_{\text{SETUP}}$ and send the values r_i, s_i in step 1.1 and $\Delta_{\gamma,r}, \Delta_{\gamma,s}, \gamma_{r,i}, \gamma_{s,i}$ 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 a_i, b_i, f_i, g_i, h_i for all \mathcal{P}_i .
 - in step 6 run SComReshare to obtain c_i .
 - Obtain the values $\Delta_{\gamma,z}, \gamma_{z,i}$ 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 a_i, f_i, b_i, g_i in step 2.2, c_i, h_i in step 2.4 and $\Delta_{\gamma,a}, \Delta_{\gamma,f}, \gamma_{a,i}, \gamma_{f,i}$ in first, $\Delta_{\gamma,b}, \Delta_{\gamma,g}\gamma_{b,i}, \gamma_{g,i}$ in the second and $\Delta_{\gamma,c}, \Delta_{\gamma,h}\gamma_{c,i}, \gamma_{h,i}$ 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

- (3) the simulator runs *Check Priples* with the adversary and the values of the honest parties. Announce 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 $e_{f,i}$ to obtain f_i
 - in step 5, it obtains (m + f)' from the adversary and (m + f) using the secret key to decrypt e_{m+f} . It then sets $\Delta_{\gamma} = (m + f)' (m + f)$
 - the simulator sets $m_1 = (m f)' f_1$ and $m_i = -f_i$ 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 $e_{f,i}$ to obtain f_i
 - in step 7, it obtains (m + f)' from the adversary and (m + f) using the secret key to decrypt e_{m+f} . It then sets $\Delta_p = (m + f)' (m + f)$
 - the simulator sets $m_1 = (m f)' f_1$ and $m_i = -f_i$ for all remaining parties

Fig. 22: Partial simulator for the offline phase, honest minority

Conversely, if $S_{\text{ONLINE,FULL}} \diamond \mathcal{F}_{\text{AUDITMPC}}$ returns REJECT on **Audit** then this happens if the simulator set $f = \bot$. 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).

Hence we see that events that trigger $\mathcal{F}_{AUDITMPC}$ 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 [18], 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 22, which as in [18] makes use of the available decryption key. A key difference is that we do provide commitments to the values of the honest parties to \mathbb{Z} , but observe that 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 open one to the correct values using the trapdoor $\log_a(h)$).

Based on Figure 22 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 3, Remark 4 and 5 as well as Lemma 2 (this is equivalent to the proof in [18] and is therefore omitted here). Moreover, the outcome of **Audit** is indistinguishable as a cheating \mathcal{A} was already caught using the zero knowledge proofs during **Compute**, and **Audit** simply also evaluates these checks.

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 κ/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, its online phase is very efficient. We note that our protocol would also work for Boolean circuits, but this would introduce a significant slowdown (since the MACs must then be defined as elements of an extension field over \mathbb{F}_2 , which leads to a significant overhead). It is an interesting future direction to design an efficient auditable protocol optimized for Boolean circuits or circuits over fields with small characteristic. With respect to online voting, there exist stronger degrees of auditability than we presented. An example is the notion of *universal verifiability* (see e.g. [34,29]) where the auditor must not know the output of the computation. We also do not provide *accountability* (see e.g. Küsters et al. [37]), and leave it as an open question whether similar, efficient protocols can be achieved in this setting.

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 [16].

Acknowledgements

We want to thank the anonymous reviewers for their helpful comments.

References

- 1. Ben Adida. Helios: Web-based open-audit voting. In USENIX Security Symposium, volume 17, pages 335–348, 2008.
- William Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In EURO-CRYPT, pages 119–135, 2001.
- 3. Miklós Ajtai, János Komlós, and Endre Szemerédi. An o(n log n) sorting network. In STOC, pages 1–9, 1983.

- 4. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In ASI-ACRYPT, pages 681–698, 2012.
- 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.
- Mihir Bellare, Juan A Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Advances in Cryptology—EUROCRYPT'98, pages 236–250. Springer, 1998.
- 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.
- 8. 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.
- 9. Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. Cryptology ePrint Archive, Report 2012/565, 2012.
- 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.
- Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation. In Advances in Cryptology—EUROCRYPT'92, pages 200–207. Springer, 1993.
- 12. David Chaum, Peter YA Ryan, and Steve Schneider. A practical voter-verifiable election scheme. Springer, 2005.
- David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM, 24(2):84–90, 1981.
- Josh D Cohen and Michael J Fischer. A robust and verifiable cryptographically secure election scheme. In FOCS, volume 85, pages 372–382, 1985.
- 15. Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. In *TRUST*, pages 55–73, 2012.
- Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In *ESORICS*, pages 1–18, 2013.
- Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.
- 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.
- 19. Sebastiaan Jacobus Antonius de Hoogh. Design of Large Scale Applications of Secure Multiparty Computation: Secure Linear Programming. PhD thesis, Technische Universiteit Eindhoven, 2012.
- 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.
- 21. 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.
- 22. Eiichiro Fujisaki and Tatsuaki Okamoto. A practical and provably secure scheme for publicly verifiable secret sharing and its applications. In *Advances in Cryptology—EUROCRYPT'98*, pages 32–46. Springer, 1998.
- Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645, 2013.
- 24. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In NDSS, 2012.
- Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer efficiently. In CRYPTO, pages 572–591, 2008.
- Alptekin Küpçü and Anna Lysyanskaya. Optimistic fair exchange with multiple arbiters. In ESORICS, pages 488–507, 2010.
- 27. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Advances in cryptology—CRYPTO'94, pages 95–107. Springer, 1994.
- Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Advances in Cryptology-ASIACRYPT 2009, pages 598–616. Springer, 2009.
- Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In CRYPTO, pages 373–392, 2006.
- Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In Proceedings of the 1st ACM conference on Electronic commerce, pages 129–139. ACM, 1999.

- 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.
- 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.
- Kazue Sako. An auction protocol which hides bids of losers. In Public Key Cryptography, pages 422–432. Springer, 2000.
- Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme. In Advances in Cryptology—EUROCRYPT'95, pages 393–403. Springer, 1995.
- Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Advances in Cryptology—CRYPTO'99, pages 148–164. Springer, 1999.
- Markus Stadler. Publicly verifiable secret sharing. In Advances in Cryptology—EUROCRYPT'96, pages 190–199. Springer, 1996.
- 37. Tomasz Truderung, Andreas Vogt, and Ralf Küsters. Accountability: definition and relationship to verifiability. In Proceedings of the 17th ACM conference on Computer and communications security, pages 526–535. ACM, 2010.

A On the Efficiency of our Solution

In this section, we will outline why the practical efficiency of the offline and audit phase of our protocol crucially depends on how fast commitments can be computed and checked. We will moreover present a few optimizations for these tasks.

Asymptotic efficiency. In terms of asymptotic efficiency, our suggested online phase is as efficient as the SPDZ protocol. Practically, the number of local field operations and sent values increases by a modest factor of two, plus some additional work for each input-providing party (to check whether the commitment is correct). It is an interesting open problem to see if one could get rid of even this minor slowdown.

To be more precise, we have to distinguish between the field operations in \mathbb{Z}_p and the group operations in G. In the standard setting, where each party provides O(1) inputs and O(1) output values are jointly computed, and where the number of gates in our circuit is upper-bounded by |C|, all operations of the online phase (**Input**, **Add**, **Multiply**, **Output**) together can be performed by each player doing at most $O(n \cdot |C|)$ field operations. Assuming that we use Pedersen commitments to implement $\mathcal{F}_{\text{COMMIT}}$ in practice, we obtain an extra $O(n \cdot \log p)$ group operations during **Input** and **Output**. In terms of network load, each party sends or receives $O(n \cdot |C|)$ field elements over the network during the **Input** phase and while the computation is carried out, and O(n) elements from \mathbb{Z}_p and G during **Output**.

We moreover have to discuss our new **Audit** phase of the protocol (we exclude $\mathcal{F}_{\text{SETUP}}$ from the discussion). The strategy of **Audit** is to follow the computation with the commitments. Here, the number of operations in \mathbb{Z}_p is $O(n \cdot |C|)$, which is comparable to the online phase. In addition, the algorithm performs the gate operations on commitments and checks whether every opening of a commitment was correct – this in total requires $O((n + C) \cdot \log p)$ group operations.

To check whether the commitments are correctly opened, the audit process computes a random linear combination of the opened commitments (using coefficients from \mathbb{Z}_p) and the values which should open them (instead of checking all of them independently). This randomized check fails with probability $2^{-\log p}$. In practice, one would choose the coefficients of the random linear combination from the smaller interval $[0, ..., 2^k - 1]$ (where we can have $k \ll \log p$), thus saving operations in G.

Towards a faster offline phase. Though the offline phase of [18] can directly be extended to support the computation of the commitments (as explained), one can use different optimizations for it. First of all, computing the commitments can be made faster using preprocessing as in [11,27]. Moreover, it is possible to reduce the total number of commitments (introducing a moderate slowdown during the online phase) as follows:

Instead of computing one commitment per value, one can 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 *i*th value. A representation $(x_1, ..., x_t, r, g_1^{x_1} \cdots g_t^{x_t} h^r)$ of t values in parallel is componentwise linear, and multiplications can also be performed as before (now for multiple elements in parallel). We observe that the computation of a commitment with many generators can be substantially faster than computing all commitments individually. This optimization, similar to [17], 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 which must then be used during the online phase. This leads to a moderate slowdown during the evaluation of the circuit.

Tweaks for the audit phase. The audit process, as explained in Figure 7, basically consists of (1) performing linear operations on commitments and (2) checking whether commitments open to the correct values. Whereas we see no approach to speed up the first part, we will address the second one using a well-known technique from [6].

Let $c_1, ..., c_n \in G$ be the commitments and let $x_1, ..., x_n, r_1, ..., r_n$ be the values that should open them. We want to establish that $\forall i \in \{1, ..., n\} : c_i = g^{x_i} h^{r_i}$.

The is to compute a random linear combination of all commitments, and thus check all of them at once. We choose the coefficients of the random combination from the interval $0, ..., 2^k - 1$.

Now computing such a random linear combination will yield a false positive with probability $\approx 2^{-k}$, but we can adjust the error probability here and make it independent of the field description size log p (remember that also G has to be a DLP-hard group of order p). This also yields less computational overhead, as we only have to raise group elements to at most 2^k th powers. The algorithm looks as follows:

(1) Choose $\boldsymbol{a} \leftarrow \{0, \dots, 2^k - 1\}^n$ uniformly at random.

(2) Check that
$$\prod_i c_i^{\boldsymbol{a}[i]} = \prod_i (g^{x_i} h^{r_i})^{\boldsymbol{a}[i]} = g^{\sum_i \boldsymbol{a}[i]x_i} h^{\sum_i \boldsymbol{a}[i]r_i}.$$

Bellare et al. show in [6] that this algorithm indeed fails to correctly verify with probability 2^{-k} . Moreover, one can use a recursive approach to gain further speedup for a large number of commitments. We refer to [6] for more details.

B A Generic Implementation of Auditable MPC

Until now, we only provided a specific implementation of $\mathcal{F}_{AUDITMPC}$ based on the SPDZ protocol. We now want to argue that it is possible to securely implement $\mathcal{F}_{AUDITMPC}$ 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 [25] or the notion of "semi-malicious" in [8]. 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 [2], 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 $\mathcal{I}_1, ..., \mathcal{I}_m$ 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 \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 $\mathcal{P}_1, ..., \mathcal{P}_n$ engage in an execution of the GMW-protocol using the stronglycorrect 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 \mathcal{T}_{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.