Cheater Detection in SPDZ Multiparty Computation

Gabriele Spini[1,2,3(✉)] and Serge Fehr[1]

[1] CWI Amsterdam, Amsterdam, Netherlands
spini@cwi.nl
[2] Mathematical Institute, Leiden University, Leiden, Netherlands
[3] Institut de Mathématiques de Bordeaux, UMR 5251, Université de Bordeaux, Bordeaux, France

Abstract. In this work we revisit the SPDZ multiparty computation protocol by Damgård et al. for securely computing a function in the presence of an unbounded number of dishonest parties. The SPDZ protocol is distinguished by its fast performance. A downside of the SPDZ protocol is that *one single dishonest party* can enforce the computation to fail, meaning that the honest parties have to *abort* the computation without learning the outcome, whereas the cheating party may actually learn it. Furthermore, the dishonest party can launch such an attack without being identified to be the cheater. This is a serious obstacle for practical deployment: there are various reasons for why a party may want the computation to fail, and without cheater detection there is little incentive for such a party not to cheat. As such, in many cases, the protocol will actually fail to do its job.

In this work, we enhance the SPDZ protocol to allow for cheater detection: a dishonest party that enforces the protocol to fail will be identified as being the cheater. As a consequence, in typical real-life scenarios, parties will actually have little incentive to cheat, and if cheating still takes place, the cheater can be identified and discarded and the computation can possibly be re-done, until it succeeds.

The challenge lies in adding this cheater detection feature to the original protocol without increasing its complexity significantly. In case no cheating takes place, our new protocol is as efficient as the original SPDZ protocol which has no cheater detection. In case cheating does take place, there may be some additional overhead, which is still reasonable in size though, and since the cheater knows he will be caught, this is actually unlikely to occur in typical real-life scenarios.

1 Introduction

The SPDZ MPC Protocol. Since the initial theoretical possibility results for multiparty computation (MPC) in the late eighties [2,5,8,10], much effort has been put into reducing the (communication and computation) complexity of MPC,

G. Spini—Supported by the Algant-Doc doctoral program, www.algant.eu.

A.C.A. Nascimento and P. Barreto (Eds.): ICITS 2016, LNCS 10015, pp. 151–176, 2016.
DOI: 10.1007/978-3-319-49175-2_8

and we are now at a stage where MPC is at the verge of being practical. One of the currently known protocols that is (close to) efficient enough for practical deployment is the so-called SPDZ protocol by Damgård et al. [7], and its variations from [6]. The efficiency of the SPDZ protocol is due to a clever mix of cryptographic operations, which can mostly be pushed into a preprocessing phase, and very efficient information-theoretic techniques.

The SPDZ MPC protocol offers security against a dishonest majority, i.e., there is no bound on the number of corrupt parties the protocol can tolerate: even if all but one of the parties are corrupt, that one single party is still protected. A downside of such protocols with security against a dishonest majority is that they are inherently susceptible to a "denial-of-service" attack: even *one single dishonest party* can enforce the protocol to fail, meaning that the honest parties have to *abort* the computation without learning the outcome, whereas the cheating party may actually learn it. Furthermore, the SPDZ MPC protocol is such that the cheating party who launches the attack remains *covert*: the (honest) parties know that there is a cheater among them that caused the protocol to fail, but they have no way to identify the culprit. As such, with little effort and with nothing to fear, a single party can prevent the SPDZ protocol from doing its job.

Identifiable vs Non-identifiable Abort. We feel that such an non-identifiable abort, where the honest parties *cannot* identify the cheating party that caused the abort, is a serious drawback for practical deployment. In real-life scenarios, there are many reasons for why a party may be tempted to enforce the protocol to fail: he may know or suspect that he is not going to like the outcome, he may gain an advantage by learning the outcome but preventing the others from learning it, he may want to sabotage the computation out of malevolence, etc. And of course, if that party does not have to fear any consequence because he knows that he will not be caught, there is little incentive for him not to cheat. As such, in real-life scenarios, it is not unlikely that such an abort will actually take place. Furthermore, once such an abort does take place, the affected honest parties are stuck — there is nothing they can do: they cannot call anyone to account, and re-trying the computation is (almost) useless, because the cheating party can just re-do the attack.

In contrast to this is the concept of *identifiable abort*, where we require that, as a consequence of launching a denial-of-service attack, the cheating party will be identified as being the culprit. Obviously, for a protocol that offers identifiable abort, there is much less incentive for a party to cheat and enforce the protocol to fail, because he knows that he will be caught and have to deal with the consequences. Thus, if there is some severe enough punishment, an abort is unlikely to occur. Furthermore, even if an abort does occur, the affected honest parties have room for further actions: not only can they call the cheating party

to account, they can also re-do the computation with the culprit excluded, and this way they can still obtain the outcome of the computation eventually.[1]

We point out that non-identifiable abort is no issue in case of *two*-party computation: if the protocol fails then it is clear to the honest party that *the other party* must be cheating.

Our Results. We propose a new version of the SPDZ protocol that supports *identifiable* abort: if the protocol aborts then at least one dishonest party will be identified as having cheated. We emphasize that the challenge lies in adding identifiability to SPDZ without increasing its complexity too much; in particular, we want the protocol to run (almost) as fast as the original version in case parties do not misbehave (too much). This is what our protocol achieves.

- In case no cheating takes place, i.e., all the players behave honesty, our protocol is essentially as efficient as the original SPDZ protocol: namely, it has an asymptotic communication complexity of $O(n)$ point-to-point communications per gate and an asymptotical computational cost of $O(n)$ field operations per gate.
  We perform extra broadcasts compared to the original SPDZ protocol, but since their number is independent of the circuit size, this can be neglected for large enough circuits.
- In case cheating does take place, but to an extent that the protocol can handle it and does not abort, our protocol is slower by a factor at most 2, hence still with an asymptotic complexity of $O(n)$ per gate for both communication and computation.
  Again, the extra broadcasts can be neglected.
- In case cheating takes place and the protocol does abort (with identification), we distinguish between the following two cases (which case occurs depends on the kind of cheating):
  - *Identification with no agreement:* Every honest player has identified at least one player as a cheater, but there may not be agreement among the honest players about the list of cheaters.[2] In this case, our protocol is slower still by a factor 2 only.
  - *Identification with agreement:* There is common agreement among the honest players about at least one player being a cheater. In this case, our protocol may take substantially longer to identify the cheater, namely in this case the number of cryptographic operations to be performed grows with the size of the circuit.

Thus, the only case when our version is significantly slower than the original SPDZ protocol is when a dishonest player cheats so bluntly that he is *publicly*

[1] One has to be careful with this "solution" though: collaborating dishonest parties that remained passive during the first run may now adjust their inputs, given that *they* have learned the output from the first (failed) run.

[2] But every player that is identified by an honest player to be a cheater *is* a cheater; thus, this case can only occur if there is more than one cheater.

recongnized as being a cheater. However, in many practical scenarios, there seems to be little gain for a dishonest player in slowing down the protocol at the cost of being publicly caught as a cheater, and thus having to face the consequences. Therefore, in typical scenarios, our protocol is similarly efficient as the original SPDZ protocol but, in contrast to the original version, it discourages dishonest players from enforcing the protocol to abort.

Related Work. Cheater detection is achieved by early MPC protocols such as [8], and by other protocols that are based on the paradigm that players prove in zero-knowledge that they followed the protocol instructions honestly. However, the high communication complexity of these protocols make them unsuitable for practical deployment.

On the other hand, recent MPC protocols (in a so-called offline/online model) are designed to have very high efficiency, like the protocols from the SPDZ family [6,7], which feature a very attractive asymptotic communication and computational complexity of $O(n)$ per multiplication gate (for the online phase). However, these protocols do not offer cheater detection. An earlier protocol by Bendlin et al. [3] offers a very weak form of cheater detection: namely, at least one honest player will identify a dishonest one, but other honest players may have no clue on the identity of cheating parties; the protocol has a computational complexity of $O(n^2)$ per multiplication gate.

The work by Ishai et al. [9] is the first to rigorously define and discuss the notion of cheater detection (in the universal-composability model of Canetti [4]); the article presents a general compiler that adds cheater detection to any semi-honest MPC protocol in the preprocessing model.

A very recent protocol, due to Baum et al. [1], builds up on the Bendlin et al. approach and achieves full cheater detection with a communication and computational complexity of $O(n^2)$ per multiplication gate; this also improves on the best protocol obtained by means of the techniques by Ishai et al.

The goal of our work is to develop a MPC protocol that is "strictly stronger" than SPDZ, in that when not under attack it has the same running time than SPDZ, and when under attack it either gives away cheaters or the protocol can handle the attack and still has the same (asymptotic) running time than SPDZ. This is achieved by our protocol, but is not achieved by any of the above. Indeed, in case no severe cheating takes place, our protocol is at most a factor 2 slower than SPDZ, hence achieving a communication and computation complexity of $O(n)$ per multiplication gate. If cheating does take place to the extent that the protocol aborts, than either we obtain a weaker notion of cheater detection ("identification with no agreement") at the same cost, or we obtain the same notion ("identification with agreement"), but with a overhead in local computations.

2 The Original SPDZ Protocol

2.1 The Setting

SPDZ allows n players $P_1, \cdots, P_n$ holding private inputs over a finite field $\mathbb{F}_q$ to securely evaluate an arithmetic circuit C on their inputs. We assume a *synchronous point-to-point communication network* that allows for perfectly private and reliable communication between any two players. We also consider a *broadcast channel*, though this one may have to be implemented using the point-to-point channels (and cryptographic techniques).

2.2 Ingredients

SPDZ follows the standard paradigm and computes the circuit C on shared values. At the core are additive sharings, for which the following notation/terminology is used.

– A $[\cdot]$-*sharing* of a value $z \in \mathbb{F}_q$ is an additive sharing of z, meaning that each player P_i holds a random *share* $z_i \in \mathbb{F}_q$ subject to $\sum_i z_i = z$. This is denoted by $[z] = (z_1, \cdots, z_n)$.

Furthermore, to ensure correctness, every shared value is accompanied by a sharing of an authentication tag for the shared value. This is formalized as follows.

– For an arbitrary but fixed $\alpha \in \mathbb{F}_q$, a $\langle\cdot\rangle_\alpha$-*sharing* of z consists of $[\cdot]$-sharings of z and of $\alpha \cdot z$, i.e., $\langle z\rangle_\alpha = \big([z], [\alpha \cdot z]\big)$. The element α is called the *global key*, and αz is called the *tag* of z and usually denoted by $\gamma(z)$. If α is clear from the context, we may write $\langle\cdot\rangle$ instead of $\langle\cdot\rangle_\alpha$.

We say that a sharing $[z]$ or a sharing $\langle z\rangle_\alpha = \big([z], [\gamma(z)]\big)$ is *privately opened* to a player P_i if each player P_j sends his share z_j to P_i via a point-to-point and P_i computes $z := \sum_j z_j$. We say that a sharing is *publicly* opened if it is privately opened to a designated "king player" P_k, and then P_k sends the reconstructed value z to all the players via point-to-point channels.[3]

Note that (for a fixed global key α) a $\langle\cdot\rangle_\alpha$-sharing is linear, in the sense that linear combinations can be computed *on the shares*:

$$\langle z + w\rangle = \langle z\rangle + \langle w\rangle := \big([z_i + w_i]_{i=1,\cdots,n},\ [\gamma(z)_i + \gamma(w)_i]_{i=1,\cdots,n}\big)$$
$$\langle \lambda z\rangle = \lambda \cdot \langle z\rangle := \big([\lambda z_i]_{i=1,\cdots,n},\ [\lambda\gamma(z)_i]_{i=1,\cdots,n}\big)\,.$$

Furthermore, if α is $[\cdot]$-shared then the same holds for addition with a constant:

$$\langle \lambda + z\rangle = \lambda + \langle z\rangle := \big([\lambda + z_1, z_2, \cdots, z_n], [\lambda\alpha_1 + \gamma(z)_1, \cdots, \lambda\alpha_n + \gamma(z)_n]\big)\,.$$

Finally, a triple $\big(\langle a\rangle_\alpha, \langle b\rangle_\alpha, \langle c\rangle_\alpha\big)$ is called a *multiplication triplet* if it consists of three $\langle\cdot\rangle_\alpha$-shared random values a, b, c subject to $ab = c$.

[3] We emphasize that, by definition, these private and public openings do *not* involve any checking of the correctness of z by means of its tag; this will have to be done on top.

2.3 Outline of the SPDZ Protocol

SPDZ is divided into a *offline* (or *pre-processing*) phase, and an *online* phase. The idea is to push most of the (somewhat) expensive cryptographic techniques into the offline phase (which can be executed *before* the inputs to the computation — or even the actual computation — are known), and rely mainly on very efficient information-theoretic techniques in the online phase.

More concretely, in the offline phase the players make use of an *additive-homomorphic* and *somehwat multiplicative-homomorphic* encryption scheme `Enc` to produce

- a $[\cdot]$-sharing $[\alpha]$ of a random and unknown global key α,
- a list of $\langle\cdot\rangle_\alpha$-sharings $\langle r\rangle_\alpha$ of random and unknown values r, and
- a list of multiplication triplets $\big(\langle a\rangle_\alpha, \langle b\rangle_\alpha, \langle c\rangle_\alpha\big)$ with random and unknown $a, b, c = ab$.

Additionally, sort of as a "side product" of the generation of all these sharings with the help of the encryption scheme `Enc`, the following is given at the end of the offline phase for every $[\cdot]$-sharing $[z] = (z_1, \ldots, z_n)$ that occurs as (first or second) component of a $\langle\cdot\rangle_\alpha$-sharing (as well as for the $[\cdot]$-sharing $[\alpha]$). Every player P_i is *committed* to his share z_i by means of an encryption $e_{z_i} := \texttt{Enc}(z_i, \rho_{z_i})$ of z_i that is publicly known, and player P_i knows the corresponding randomness ρ_{z_i}. Recall that `Enc` is additively-homomorphic, so that linear combinations (and addition with constants) can be computed on the commitments.

The actual computation takes place in the online phase. By using the sharings produced in the offline phase as a resource, the online phase can be executed to a large extent by means of very efficient information-theoretic techniques — the number of cryptographic operations needed is independent of the circit size. Concretely, the online phase is composed of the following gadgets.

- *Input sharing:* For each input x held by a player P_i, a fresh (meaning: yet unused) sharing $\langle r\rangle_\alpha$ from the offline phase is selected and privately opened to P_i. P_i then sends $\varepsilon := x - r$ to all the players, and altogether the players can then compute a sharing of x as $\langle x\rangle_\alpha = \varepsilon + \langle r\rangle_\alpha$.
- *Distributed addition (and multiplication/addition with constants):* For each addition gate in the circuit with shared inputs $\langle z\rangle_\alpha$ and $\langle y\rangle_\alpha$, a sharing of $z+y$ is computed (non-interactively) as $\langle z + y\rangle_\alpha = \langle z\rangle_\alpha + \langle y\rangle_\alpha$. Correspondingly for multiplication/addition with a constant.
- *Distributed multiplication:* For each multiplication gate in the circuit with shared inputs $\langle z\rangle_\alpha$ and $\langle y\rangle_\alpha$, a sharing of $z \cdot y$ is computed (interactively) by means of the multiplication subprotocol below, which consumes one fresh multiplication triple from the offline phase.
- *Output reconstruction*: For each shared output value $\langle z\rangle_\alpha$, the players publicly reconstruct z.

– *Tag checking:* For a shared value $\langle z \rangle_\alpha = ([z], [\gamma(z)])$ that has been publicly opened, the players can check the correcntess of z as follows. Every player P_i computes $y_i := \gamma(z)_i - z \cdot \alpha_i$ and broadcasts a commitment of y_i, and then every player opens the commitment and the players compute $y := \sum_i y_i = \gamma(z) - z \cdot \alpha$. If $y = 0$ then z is declared to be correct; otherwise, it is declared incorrect and the protocol is immediately *aborted*.

We do not detail how these gadgets are put together, in particular how/when exactly the *tag checking* is used, as this is not very relevant to us. However, let us emphasize that a single dishonest player can easily enforce the protocol to abort, e.g., by submitting an incorrect share for a sharing $\langle z \rangle_\alpha$ that is publicly opened and then checked; the check will recognize (with high probability) that the reconstructed value z is incorrect, and so the protocol will abort, but there is no way for the honest players to find out *who* submitted an incorrect shares. Hence, any such dishonest player gets away with it, and hence there is no incentive for a dishonest player *not* to cheat, should it give him any advantage or satisfaction whatsoever.

Multiplication subprotocol
A fresh multiplication triplet $\big(\langle a \rangle, \langle b \rangle, \langle c \rangle\big)$ is selected, and the following is performed.

1. The players compute $\langle \varepsilon \rangle := \langle z - a \rangle$ and $\langle \delta \rangle := \langle y - b \rangle$.
2. The sharings $\langle \varepsilon \rangle$ and $\langle \delta \rangle$ are publicly opened:
   – $\langle \varepsilon \rangle$ and $\langle \delta \rangle$ are privately opened to a designated king player P_k, and
   – P_k sends ε and δ to the others player via the point-to-point channels.
3. The players compute $\langle z \cdot y \rangle := \langle c \rangle + \varepsilon \langle b \rangle + \delta \langle a \rangle + \varepsilon\delta$.

3 Our Protocol

3.1 An Overview of Our Protocol

We explain on a high level how our protocol works. First, notice that there are three distinct ways for dishonest players to disrupt the protocol execution (and enforce an abort in the original SPDZ protocol):

– During the input sharing phase, dishonest players could send incorrect shares of r to P_i, or P_i could send inconsistent values ε to the players.[4]
– During the multiplication step, dishonest players could send incorrect shares of ε and δ to the king player.

[4] Note that there is no issue of ε being *incorrect* since any ε corresponds to a possible input for P_i.

– During the multiplication step, a dishonest king player could send false and/or inconsistent values for ε and δ.
– In the output reconstruction phase, dishonest players could announce false shares of the output.

We will focus on the two possible attacks in the multiplication step, since our techniques to deal with those can easily be used to also deal with the attacks in the input sharing and output reconstruction phases.

As pointed out above, the players have two "checking mechanisms" available in order to verify the correctness of a reconstructed value z:

– they can use the *tag* $\gamma(z)$ of z to check the correctness of z, and
– they can use the *commitments* to check the correctness of the shares z_i.

The former technique is very efficient but cannot be used to identify *who* submitted a false share in case of an incorrect z; this can be done by the latter, but that one is computationally more expensive, and so we want to avoid it as much as possible and use it only as a "last resort".

Now, a first and straightforward approach to achieve cheater detection but use the computationally expensive techniques only as a last resort, seems as follows: first, use the "cheap" tag checks to verify the correctness of every reconstructed value (as in the original SPDZ protocol), and then resort to the commitments if and only if an error is detected, in order to find out who claimed an incorrect value.

Unfortunately, this does not work. The reason is that only the king player knows the shares of, say, ε. As such, if ε claimed by the king player turns out to be incorrect, there is no way for an honest player to distinguish the case of a dishonest player P_i who has sent an incorrect share ε_i to the king player, from the case of a dishonest king player who pretends that he has received an incorrect share ε_i from P_i. There is no way such a dispute can be resolved, even with the help of the commitments — except if these shares are broadcast from the start, but that would greatly increase the complexity of the protocol.

To deal with such a situation, we use an idea from *dispute control*: we re-do (part of) the computation in such a way that this particular dispute cannot occur anymore (essentially by choosing a fresh king player). Since the number of disputes is bounded, this means that there is a limit on how often something needs to be re-done, and setting the parameters right ensures that this merely gives a factor-2 blowup.

On the other hand, if a dishonest player P_i keeps on claiming an incorrect share for, say, ε, even when the players are asked to *broadcast* their shares because a fault was detected, then the players can use the (computationally expensive) commitments to find the incorrect share, and the honest players will unanimously identify P_i as cheater.

The overall structure of (the computation phase of) our protocol is thus as follows.

Set-up: The circuit C is divided into consecutive blocks, each comprising ca. $|C|/n$ gates (where "consecutive" here means that C can be evaluated in a block-by-block manner). Furthermore, a list $\mathcal{L}_{\texttt{suspects}}$ of *suspect* players is initialized as the empty set.

Computation: Sequentially, for each block the following is done:

I. A king player $P_k \notin \mathcal{L}_{\texttt{suspects}}$ is selected, and the computation is done as in the normal SPDZ protocol by repeatedly invoking the multiplication sub-protocol and doing local computations.

II. Once the block has been processed, a checking protocol `BlockCheck` is invoked that verifies the correctness of the computation. `BlockCheck` has three possible outcomes:
   - *Success:* The block has been correctly processed. In this case, the players simply move to the next block.
   - *Fail with Conflict:* The block has *not* been correctly processed, and P_k accuses some player(s) of faulty behaviour. In this case, P_k and all accused players are added to $\mathcal{L}_{\texttt{suspects}}$, and the players go back to step I. and re-do the computation with a "fresh" $P_k \notin \mathcal{L}_{\texttt{suspects}}$. Should $\mathcal{L}_{\texttt{suspects}}$ now consist of *all* players then the protocol stops; in this case, every honest player has identified at least one dishonest player.
   - *Fail with Agreement:* The block has *not* been correctly processed, and it is guaranteed that some player has *broadcast* an incorrect share during the run of `BlockCheck`. In this case, the players make use of the commitments to unanimously identify the cheating player.

3.2 The Checking Protocol `BlockCheck`

We will now provide a more precise discussion of the check-phase mentioned in the previous section. What it will do is check the correctness and consistency of all the ε's and δ's that were announced by the king player during the multiplication subprotocols in the block to be checked. Let us write $\langle z^{(1)} \rangle, \cdots, \langle z^{(t)} \rangle$ for the sharings of these ε's and δ's. This means that each player P_i has communicated his share $z_i^{(j)}$ to P_k, who in turn has computed $z^{(j)} = \sum_i z_i^{(j)}$ and communicated it to all other players. In the following discussion, we will denote by $\tilde{z}_i^{(j)}$ the *actual* share communicated by P_i to P_k (so that if P_i is dishonest, it may be the case that $\tilde{z}_i^{(j)} \neq z_i^{(j)}$), and by $\tilde{z}^{(j)}$ the value that P_k has communicated to the other players. However, we emphasize that that P_k is dishonest then he may be inconsistent with the value of $\tilde{z}^{(j)}$, different players may receive different values for $\tilde{z}^{(j)}$.

The subprotocol `Block Check` now works as follows. As a first step, as checking each value individually is too expensive, a quasi-random linear combination of the $\langle z^{(j)} \rangle$'s is computed:

> *Step 1:* The players run a subroutine `Rand` to produce a random element e, and they compute the linear combination $\langle z \rangle := \sum_h e^h \langle z^{(h)} \rangle$.

Let $\tilde{z}_i$ and $\tilde{z}$ be the respective linear combinations of $\tilde{z}_i^{(1)}, \cdots, \tilde{z}_i^{(t)}$ and of $\tilde{z}^{(1)}, \cdots, \tilde{z}^{(t)}$. The correctness of $\tilde{z}$ is then verified as follows.

> *Step 2: Public Reconstruction.* Each player broadcasts his share of $\langle z \rangle$, upon which the king player P_k broadcasts a list of players that he accuses of inconsistent behaviour; if he does so, `BlockCheck` outputs the message "Fail with Conflict" and the list of accused players.
>
> If P_k has not accused anybody, then each player can broadcast an accusation against P_k, stating that the value $\tilde{z}$ that he received is different from z (which is now public, since its shares have been broadcast). If that is the case, then once again we are in the "Fail-with-Conflict" case: `BlockCheck` outputs the corresponding error message and the list of players accusing P_k.

Now if no accusations have been produced, the next step consists in checking the tag of the now-public value $\tilde{z}$:

> *Step 3: Tag Checking.* If no accusations have been produced in Step 2, then players check the tags of $\langle z \rangle = ([z], [\gamma(z)])$; this is done running a subroutine `ZeroTest` on $[\gamma(z)] - \tilde{z}[\alpha]$, which outputs $\top$ if it is a sharing of 0, and $\bot$ if it is not (except with small probability).
>
> `BlockCheck` outputs the message "Success" if the tag check succeeds, and "Fail with Agreement" if it fails.

Note that in step 3 above, the players cannot just do a public reconstruction of the sharing $[\gamma(z)] - \tilde{z}[\alpha]$ to check whether it is a sharing of zero, because in case it is not, the value of $\gamma(z) - \tilde{z}\alpha$ reveals information on α. That is why a slightly more involved subroutine `ZeroTest` is invoked, which publicly reconstructs a *random multiple* of $[\gamma(z)] - \tilde{z}[\alpha]$.

If the tag check in step 3 above fails, then this means that a dishonest player P_i must have broadcast a false share z_i during step 2, *or* (as we will see) he has broadcast some false share as part of the execution of `ZeroTest`; in either case, he has broadcast a linear combination (with coefficients that may depend on the $\tilde{z}^{(i)}$) of values he is committed to, by means of the commitments from the preprocessing phase and by the linearity of all computations. P_i can and will now be *publicly identified* as cheater by means of a protocol `CommitCheck`,

which simply asks the players to open the commitments to the claimed and broadcast values. We note that `CommitCheck` causes a significant overhead to the efficiency of the protocol because the players need to perform computations on a large number of commitments, proportional to the size of the circuit; in return, however, it publicly identifies a cheating player.

Note that due to space contstraints, the details of the subroutines `Rand`, `CommitCheck` and `ZeroTest` are given in the appendix, but they are pretty much as expected (except for the issue mentioned above regarding `ZeroTest`).

3.3 Security of Our Protocol

In this section we argue security of our protocol. We focus on the actual computation phase; similar techniques allow us to secure the input-sharing and output-reconstruction phases as well.

The security of the protocol clearly relies on the secrecy of the global key α, which we measure as follows. Let v denote the adversary's *view* at a given point in the online protocol.[5] Then, the adversary's *(average) guessing probability* of the global key α is given by

$$p_{\texttt{guess}}(\alpha|v) := \sum_{\hat{v}} p(v = \hat{v}) \cdot \max_{\hat{\alpha}} p(\alpha = \hat{\alpha}|v = \hat{v}) \,.$$

In Appendix A we will prove the following security properties for the checking protocol `BlockCheck`. Recall, the purpose of `BlockCheck` is to verify the correctness and consistency[6] of $\tilde{z}^{(1)}, \cdots, \tilde{z}^{(t)}$, which is the collection of values that P_k announces as the reconstructed values for $\langle \epsilon \rangle$ and $\langle \delta \rangle$ for each invocation of the multiplication subprotocol in the checked block.

Proposition 1. *`BlockCheck` satisfies the following:*

- Correctness of `BlockCheck`*: if all players behave honestly and hence all $\tilde{z}^{(h)}$ are correct and consistently announced by P_k, then `BlockCheck` outputs "Success" with probability 1.*
- Soundness of `BlockCheck`*: if at least one of the $\tilde{z}^{(h)}$ is incorrect, i.e. $\neq z^{(h)}$, or inconsistently announced by P_k, then the following holds except with probability at most*

$$\delta = (2|C|/n + 1)/q + p_{guess}(\alpha|v) \,,$$

where v is the adversary's view before the execution of `BlockCheck`.
`BlockCheck` outputs a "Fail"; furthermore, if it outputs "Fail with Conflict",

[5] Here and below, when we make information-theoretic statements, we understand v to *not* include the encryptions/commitments of the honest parties shares etc. that were produced during the preprocessing phase. Adding these elements to the adverary's view of course renders the information-theoretic statements invalid, but has a negligible effect with respect to a computationally bounded adversary.

[6] Recall that dishonest P_k may send different values for $\tilde{z}^{(i)}$ to different players.

then either the king player P_k or all of the accusing players are dishonest (or both), and if it outputs "Fail with Agreement", then all $\tilde{z}^{(h)}$ have been consistently announced by P_k, and a dishonest player has broadcast as part of `BlockCheck` *an incorrect version of a value to which he is committed by means of a linear combination (depending on the $\tilde{z}^{(h)}$'s) of the commitments produced in the preprocessing phase.*

Notice that for the above soundness error to be small, we need to bound $p_{\texttt{guess}}(\alpha|v)$. Clearly, at the beginning of the online phase it is $1/q$, but it may increase during the course of the protocol. We have the following upper bound, which will be proved in Appendix A:

Proposition 2. *Throughout the entire protocol, the adversary's guessing probability of α is bounded by*

$$p_{guess}(\alpha|v) \leq \frac{1}{q-2n} + \frac{2n}{q}.$$

Finally, for completeness, we state here that `CommitCheck`, which will be invoked if `BlockCheck` results in "Fail with Agreement", does the job and identifies a dishonest player. Crucial for `CommitCheck` to work properly is that the $\tilde{z}^{(h)}$ have been consistently announced by P_k (so that there is agreement on the linear combination to be computed on the commitments), but this is ensured by the soundness of `BlockCheck`.

Proposition 3. *Under the binding property of the underlying commitment scheme, if a dishonest player has broadcast as part of* `BlockCheck` *an incorrect value, then this player will be publicly identified by* `CommitCheck`*. Furthermore, no honest player will incorrectly be identified as being dishonest.*

The security of the protocol is now straightforward: as a worst-case scenario, we will assume that the adversary controls all but one of the players. First notice that if the adversary decides to behave (semi)-honestly, then by the correctness of `BlockCheck` the protocol will reach the end of the circuit and `CommitCheck` will not be executed.

On the other hand, if the adversary misbehaves in (at least) one of the invocations of the multiplication subprotocol in one of the blocks, either by sending an incorrect share of $\langle\varepsilon\rangle$ or $\langle\delta\rangle$ to P_k, or by having dishonest P_k announce inconsistent values (or both), then this will be detected by `BlockCheck` that will announce "Fail with Conflict" or "Fail with Agreement", depending on the adversary's precise behavior.

In the case of a "Fail with Conflict", the incorrect data is dismissed and the block is rebooted with a fresh king player that is not in the list $\mathcal{L}_{\texttt{suspects}}$ of suspect players. Since every re-boot adds a new player to $\mathcal{L}_{\texttt{suspects}}$, namely the previous king player, we can have at most n such reboots in total before the protocol produces the correct output or before $\mathcal{L}_{\texttt{suspects}}$ is "full", and in that case the protocol stops and every honest player has correctly identified at least one dishonest player (because an honest player ends up in $\mathcal{L}_{\texttt{suspects}}$ only by accusing

a dishonest player). Notice that the commitment check is not executed in this case. On the other hand, if `BlockCheck` ends with a "Fail with Agreement", then `CommitCheck` is invoked and will publicly identify a dishonest player.

As for the overall error probability, by combining the soundness error of `BlockCheck` with the bound on $p_{\texttt{guess}}$, and observing that `BlockCheck` is invoked at most $2n$ times — as we have n blocks plus at most n reboots — we obtain an overall error probability of at most

$$\varepsilon = 2n \cdot \left(\frac{1}{q - 2n} + \frac{2|C|/n + 2n + 1}{q} \right)$$

To sum up, and using a similar checking mechanism for ensuring correctness of the input-sharing and the output-reconstruction phases, our new multiparty computation protocol satisfies the following.

Theorem 1. *For any computationally bounded adversary that cannot break the encryptions/commitments used in the preprocessing phase, except with negligible probability, an execution of our protocol results in one of the following cases (depending on the adversary's strategy):*

I. Success: *the protocol reaches the end of the circuit and outputs the correct result to all players. In this case,* `CommitCheck` *is not executed.*
II. Identification without agreement: *the protocol aborts, but each honest player has identified at least one dishonest player. Also in this case,* `CommitCheck` *is not executed.*
III. Identification with agreement: *the protocol aborts, and the honest players have in-agreement identified at least one dishonest player.*

Furthermore, in all cases, the adversary learns no information on the honest players' inputs, beyond the result of evaluating the circuit C on the inputs.

3.4 The Complexity of Our Protocol

We discuss in this section the complexity of our protocol; as with the previous sections, we focus on the multiplication check, which is the most expensive part of our protocol. The input sharing and the output reconstruction, moreover, can be analyzed in a similar fashion (i.e., in the general case they yield a complexity of the same order of magnitude as the original SPDZ, and exceed it only to unanimously identify a dishonest player).

We thus focus on the multiplication check. We first study the complexity of processing and checking a single block:

– First, the gates of the block are evaluated as in standard SPDZ; this yields a complexity of $|C|/n \cdot O(n) = O(|C|)$ field operations (in total over all players) and the same number of field elements for point-to-point communication, and no broadcasts.

– At the end of the block, the subprotocol `BlockCheck` is executed. Its computation complexity is dominated by computing the linear combination of $t = 2 \cdot |C|/n$ sharings on n shares in step 1; in total, as we will see in Appendix A.2, `BlockCheck` has a computation complexity of $O(|C| + n^2)$ field operations, plus preparing $4n$ commitments. Its communication complexity consists of no point-to-point communication, and $3n$ broadcasts of field elements and $4n$ of commitments and openings.

Now as we have seen, `BlockCheck` can lead an execution of `CommitCheck`, to a re-boot of the current block, or simply to the processing of the following block. The exact cost of `CommitCheck` depends on how the commitments were implemented, but it certainly causes a significant overhead given that it involves a number of cryptographic operations that grows with the size of the circuit; however, `CommitCheck` leads to the public exposure of a dishonest player, so there is little incentive for the adversary to enforce this. As argued in Sect. 3.3, we can have at most n reboots in total before the protocol aborts; as such, the overhead of the reboots causes at most a factor 2 overhead to the ordinary computation of the n blocks.

We thus get the following result summarizing the complexity of our protocol:

Proposition 4. *Except in the case where* `CommitCheck` *is enforced by the adversary, which would lead to an identification-with-agreement of at least a dishonest player, our protocol has the following complexity:*

– *Computation:* $O(n|C| + n^3)$ *field operations, plus preparing* $8n^2$ *commitments (as part of* `Rand` *and of* `ZeroTest`*);*
– *Communication:* $O(n|C|)$ *field elements for point-to-point communication plus* $O(n^2)$ *broadcasts.*[7]

In case `CommitCheck` *is executed, the communication complexity remains the same, while players need to execute* $O(n^2|C|)$ *cryptographic operations on top of the original computational complexity.*[8]

Compared to the original SPDZ protocol, in case all players behave honestly, our protocol is as efficient as the original protocol, up to an additive overhead caused by an increased number of commitments and broadcasts[9], but this overhead is independent of the circuit size and thus negligible except for small circuits. In case of active cheating — unless a dishonest player cheats so bluntly that

[7] Note that we treat broadcast as a given primitive here; implementing it using the point-to-point communication and, say, digital signatures, causes some (communication and computation) overhead, but this overhead is independent of the circuit size.

[8] The actual cost of these cryptographic operations depends on how the commitment scheme is implemented.

[9] Plus that we have to do *real* broadcasts, whereas in the original SPDZ protocol without cheater detection it is good enough to do a simple consistency check and abort as soon as there is an inconsistency.

`CommitCheck` is invoked and he will be publicly identified as being a cheater — the (computation and communication) complexity of our protocol is larger by a factor 2 only, plus the same kind of additive overhead that does not depend on the circuit size.

4 Conclusion

We presented an alternative to the original SPDZ multiparty computation protocol. In contrast to the original protocol, our version allows for *cheater detection*. As such, our protocol is much less vulnerable to a "denial of service" attack: if a dishonest player enforces the protocol to abort, he will be identified and measures can be taken. Furthermore, in our protocol, this feature comes essentially *for free*: as long as everything works as supposed, our protocol is *as efficient* as the original SPDZ (up to an additive overhead that is negligible except for small circuits); but as soon as a fault is detected, instead of simply aborting and being clueless about who cheated, we can proceed and — depending on the adversary's behavior — still *complete the computation* or *identify cheaters without agreement* with a factor 2 overhead, or *identify cheaters with agreement* but with a significant overhead.

As such, we think that our multiparty computation protocol is an attractive alternative to the original SPDZ protocol when considering real life scenarios where dishonest parties may have various incentives for sabotaging an execution.

An obvious open problem is to have agreement on the cheater(s) *in all cases*, and/or without a significant overhead; however, this seems hard to achive without increasing the complexity of the honest execution.

A The Protocol `BlockCheck` in Detail

We shall now begin the study of the sub-protocol `Block Check`; we first establish some notation rules that will be used in the whole section: t will denote a positive integer; we assume that t multiplication opening values $\langle z^{(1)} \rangle, \cdots, \langle z^{(t)} \rangle$ have been publicly opened via a king player P_k, and we will use the following notation: for each shared value $\langle z^{(h)} \rangle$,

- each player P_j has sent $z_j^{(h)}$ to P_k;
- $\tilde{z}_j^{(h)}$ denotes the value received by P_k from P_j (so if P_j is honest, $\tilde{z}_j^{(h)} = z_j^{(h)}$);
- P_k has computed and sent to each P_j the value $z^{(h)}$;
- $\tilde{z}^{(h)}(j)$ denotes the value received by each P_j from P_k (so if P_k is honest, $\tilde{z}^{(h)}(j) = z^{(h)}$).

The goal of `BlockCheck` is to detect errors in this process; as we have seen, the first step of the check consists in computing a (quasi-) random linear combination of the values to be checked. This is performed by generating a seed via the subroutine `Rand`, and then using the powers of the seed as coefficients of the linear combination. We first define `Rand`, which assumes that players have access to a commitment scheme (as in standard SPDZ):

`Rand`:
The protocol is used to generate a random seed $e \in \mathbb{F}_q$.

(i) Each player P_j selects random $e_j \leftarrow \mathbb{F}_q$ and broadcasts a commitment $\texttt{Commit}(e_j)$ to it;
(ii) all the commitments are then opened, so that all players get $e_1, \cdots, e_n$;
(iii) the output of `Rand` is the value $e := \sum_{j=1}^{n} e_j$.

We now show that any error that occurred during the opening of the values $\langle z^{(1)} \rangle, \cdots, \langle z^{(t)} \rangle$ will affect their linear combination as well (with high probability); the proof is a standard argument, and is omitted here. We refer to the full version of the paper for the details.

Lemma 1. *Let e be a seed generated by `Rand`; consider the following linear combination with coefficients given by the powers of e:*

$$\langle z \rangle := \sum_{h=1}^{t} e^h \cdot \langle z^{(h)} \rangle, \quad \tilde{z}(j) := \sum_{h=1}^{t} e^h \cdot \tilde{z}^{(h)}(j) \text{ for any } j = 1, \cdots, n.$$

Assume that for a given index $h \in \{1, \cdots, t\}$ the value received by a given player P_j is incorrect, i.e. $\tilde{z}^{(h)}(j) \neq z^{(h)}$; then $\tilde{z}(j) \neq z$ except with probability t/q.

Similarly, if the values received by two players P_j and P_i for an index h are different (i.e. $\tilde{z}^{(h)}(j) \neq \tilde{z}^{(h)}(i)$), then the same will hold for the corresponding linear combinations, i.e. $\tilde{z}(j) \neq \tilde{z}(i)$ except with probability t/q.

The next step of `BlockCheck` is the "public opening and conflict" phase; it has already been defined in previous sections, but we will re-write it here in order to make this chapter as self-contained as possible:

`PublicOpening`:
The protocol takes as input a shared value $[z]$ and the index k of the king player P_k; initialize the boolean value $\mathbf{b}$ to $\top$ and the list L to the empty set $\emptyset$.

(i) For each $j = 1, \cdots, n$, player P_j broadcast z_j and P_k broadcast $\tilde{z}_j$; if the two values do not coincide, set $\mathbf{b} = \bot$ and $L \leftarrow L \cup \{P_j\}$.
(ii) If $\mathbf{b} = \bot$, the protocol stops and output $(\bot, L)$.
(iii) Players set $\tilde{z} := \sum_j z_j$; for each $j = 1, \cdots, n$, player P_j broadcasts $\tilde{z}(j)$. If this value is different from $\tilde{z}$, set $\mathbf{b} = \bot$ and $L \leftarrow L \cup \{P_j\}$.
(iv) The protocol outputs $(\mathbf{b}, L, \tilde{z})$.

The following lemma is a direct consequence of the definition of the algorithm, and states that the public opening routine is correct and sound:

Lemma 2. *Let* $(\mathbf{b}, L, \tilde{z})$ *be the output of* `PublicOpening`$([z])$ *with king player* P_k*; we then have the following properties:*

- (correctness) *if* $\langle z \rangle$ *has been correctly reconstructed and players follow the instructions of the protocol, then* $\mathbf{b} = \top$ *and* $L = \emptyset$*.*
- (soundness) *if* $\tilde{z}(j) \neq \tilde{z}(i)$ *for some honest players* P_j *and* P_i*, then* $\mathbf{b} = \perp$ *and* $L \neq \emptyset$*.*
  Furthermore, in this case either P_k *or all players in* L *are dishonest.*

The last step consists in checking the tags of the value $\langle z \rangle = ([z], [\gamma(z)])$; as we have previously discussed, this is performed by using the subroutine `ZeroTest` to check that the value $[\gamma(z)] - \tilde{z}[\alpha]$ opens to 0.

As hinted in Sect. 3.2, we need to be careful when checking the tags via `ZeroTest`, as this can increase the adversary's guessing probability of α. We introduce the following definition to model the information on α possessed by the adversary:

Definition 1. *Given a distribution* $p(x, v)$*, we say that the distribution of* x *given* v *is* a list of size m *if there exists a (conditional) distribution* $p(\ell|v)$*, where the range of* ℓ *consists of lists of* m *elements in the range of* x*, such that the following two properties hold for the joint distribution* $p(x, v, \ell) := p(x, v) \cdot p(\ell|v)$*:*

(I) $p(x \in \ell) \leq \max_{\hat{\ell} \in Im(\ell)} p(x \in \hat{\ell})$*;*

(II) $p(x|v = \hat{v}, \ell = \hat{\ell}, x \notin \hat{\ell}) = p(x|x \notin \hat{\ell})$ *for every* $\hat{v}$*,* $\hat{\ell}$ *such that the formula is well-defined.*

In a nutshell, we use the above definition to formalize the following situation: let v denote the adversary's view and $x := \alpha$; assume that the distribution of α given v is a list of size m. This means that the adversary has tried to guess the value of α for m consecutive times, and he has learned whether his guess was correct or not after each guess.

We now state the basic properties of `ZeroTest`, which will in turn imply the desired properties of the tag check; we assume that `ZeroTest` outputs a boolean value $\mathbf{b} \in \{\top, \perp\}$, marking whether the input opens to zero or not, and some extra data that will be omitted in the following lemma.

Lemma 3. *Let* $\mathbf{b}$ *be the output of* `ZeroTest`$([x])$*; we then have the following properties:*

- (correctness)*: if* $x = 0$ *and players follow the instructions of the protocol, then* $\mathbf{b} = \top$ *with probability 1.*
- (soundness)*: consider the joint distribution* $p(x, v_0)$*, where* v_0 *denotes the adversary's view before the execution of* `ZeroTest`*. Then*

$$p(\mathbf{b} = \top) \leq 1/q + p_{guess}(x|v_0).$$

Furthermore, if $x = 0$ *but* $\mathbf{b} = \perp$*, then a dishonest player has broadcast an incorrect version of a value to which he is committed by means of a linear combination of the commitments produced in the preprocessing phase.*

– (privacy)*: Assume that x is uniformly distributed and that the distribution of x given v_0 is a list of size m_0. Then after the execution of* `ZeroTest`$([x])$*, the distribution of x given v is a list of guesses of size at most $m := m_0 + 1$, where v denotes the adversary's view after the execution of* `ZeroTest`*.*

Now that we have fixed the notation for the subroutines, we can state the definition of `BlockCheck` in a more formal way:

`BlockCheck`:
The protocol takes as input a block and the index k of the king player P_k; denote by $\langle z^{(1)} \rangle, \cdots, \langle z^{(t)} \rangle$ the multiplication opening values of the block.

(i) Players execute `Rand` to get a random seed $e \in \mathbb{F}_q$, then compute the linear combination $\langle z \rangle := \sum_{h=1}^{t} e^h \langle z^{(h)} \rangle$.
(ii) Run $\left(\mathbf{b}, L, \tilde{z}\right) \leftarrow$ `PublicOpening` $([z])$; if $\mathbf{b} = \bot$, `BlockCheck` stops and outputs the message "Fail with Conflict" together with the list L.
(iii) Run $\left(\mathbf{b}, (\langle a \rangle, \langle b \rangle, \langle c \rangle), \langle r \rangle\right) \leftarrow$ `ZeroTest` $([\gamma(z)] - \tilde{z}[\alpha])$.
(iv) If $\mathbf{b} = \top$, output the message "Success";
if $\mathbf{b} = \bot$, output the message "Fail with Agreeement" together with the elements $\left((\langle a \rangle, \langle b \rangle, \langle c \rangle), \langle r \rangle\right)$.

We can now prove the properties of `BlockCheck` claimed in Sect. 3.3; we omit the proof here, as it can be easily derived from the definition of `BlockCheck`.

Proposition 5. *`BlockCheck` satisfies the following:*

Correctness of `BlockCheck`*: if all players behave honestly and hence all $\tilde{z}^{(j)}$ are correct and consistently announced by P_k, then* `BlockCheck` *outputs "Success" with probability 1.*

Soundness of `BlockCheck`*: if at least one of the $\tilde{z}^{(j)}$ is incorrect, i.e. $\neq z^{(j)}$, or inconsistently announced by P_k, then the following holds except with probability at most*

$$\delta = (2|C|/n + 1)/q + p_{guess}(\alpha|v)\,,$$

where v is the adversary's view before the execution of `BlockCheck`*.* `BlockCheck` *outputs "Fail"; furthermore, if it outputs "Fail with Conflict", then either the king player P_k or all of the accusing players are dishonest (or both), and if it outputs "Fail with Agreement", then all $\tilde{z}^{(j)}$ have been consistently announced by P_k, and a dishonest player has broadcast as part of* `BlockCheck` *an incorrect version of a value to which he is committed by means of a linear combination (depending on the $\tilde{z}^{(j)}$'s) of the commitments produced in the preprocessing phase.*

Finally, we can now prove the bound on the adversary's guessing probability of the global key α:

Proposition 6. *Throughout the entire protocol, the adversary's guessing probability of α is bounded by*

$$p_{guess}(\alpha|v) \leq \frac{1}{q - 2n} + \frac{2n}{q}.$$

Proof. Clearly, the adversary can increase his guessing probability only during the execution of `ZeroTest`; this, by definition of `BlockCheck`, is executed only when the value $\tilde{z}$ is consistent among players, so that its input is equal to $[x] := [\gamma(z)] - \tilde{z}[\alpha] = (z - \tilde{z})[\alpha]$. Hence we can assume as a worst-case scenario that $z \neq \tilde{z}$, so that the adversary's guessing probabilities of α and of x coincide.

Notice that at the beginning of the computation, the distribution of α given the adversary's view is a list of guesses of size 0; hence we can inductively apply Lemma 3, so that during the execution of the protocol the distribution of α given the adversary's view is a list of guesses of size at most $2n$ (recall that `BlockCheck`, and hence `ZeroTest`, is executed at most $2n$ times). Hence according to Definition 1, and given that α is uniformly distributed, there exists a distribution $p(\ell|v)$ with the following properties:

(I) $p(\alpha \in \ell) \leq 2n/q$;
(II) $\max_{\hat{\alpha},\hat{\ell}} p(\alpha = \hat{\alpha}|v = \hat{v}, \ell = \hat{\ell}, \alpha \notin \hat{\ell}) = 1/(q-m)$.

Now from this we can deduce the claimed upper bound on the guessing probability: indeed, by using the law of total probability with the events $(\alpha \in \ell)$ and $(\alpha \notin \ell)$, we obtain

$$p_{\mathtt{guess}}(\alpha|v) = \sum_{\hat{v}} p(v = \hat{v}) \cdot \max_{\hat{\alpha}} p(\alpha = \hat{\alpha}|v = \hat{v}) \leq \frac{1}{q - 2n} + \frac{2n}{q}\,.$$

□

A.1 The Tag Checking in Detail

We discuss in this section the sub-routine `ZeroTest`, meant to check whether some shared value $[x]$ is equal to zero or not. The key point is that we cannot simply open $[x]$: indeed, in the actual scenario this value will be equal to $[\gamma(z)] - \tilde{z}[\alpha]$ for some shared value $\langle z \rangle$; now the adversary could select any value Δz and let $\tilde{z} = z + \Delta z$, so opening $[\gamma(z)] - \tilde{z}[\alpha] = \Delta z \cdot [\alpha]$ will actually let the adversary learn the global key α. This is not a problem in the original SPDZ protocol, since it will abort if the value does not open to 0, but it is a problem for our protocol, which carries on even if the result is not zero. To avoid this, we will perform a multiplication of $[x]$ with a random shared value:

`ZeroTest`:
The protocol takes as input a shared value $[x]$.

(i) Players select a random shared value $\langle r \rangle$ and a fresh multiplication triplet $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$.
(ii) Players compute $[rx]$ with multiplication triplet $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ as described in Section 2, but with a different communication model: instead of sending their data to a king player that acts as a relay, they will broadcast a commitment to it, then open all the commitments before moving to the next round.

(iii) Each player P_j broadcasts a commitment $\texttt{Commit}((rx)_j)$ to his share of $[rx]$, then all commitments are opened, so that players obtain rx.
(iv) Set $\mathbf{b} = \top$ if $rx = 0$, $\mathbf{b} = \bot$ otherwise; output $\big(\mathbf{b}, (\langle a\rangle, \langle b\rangle, \langle c\rangle), \langle r\rangle\big)$.

From now on, we will adopt a slight abuse of notation by writing formulae such as $\texttt{ZeroTest}([x]) = \mathbf{b}$, i.e. considering only the boolean value among the outputs of the protocol. We first prove that the subprotocol is correct and sound:

Lemma 4. *`ZeroTest` satisfies the following properties:*

- Correctness: *if players follow the instructions of the protocol,* $\texttt{ZeroTest}([0]) = \top$ *with probability 1.*
- Soundness: *consider the joint distribution* $p(x, v_0)$*, where* v_0 *denotes the adversary's view before the execution of* `ZeroTest`*; then*

$$p(\texttt{ZeroTest}([x]) = \top) \leq 1/q + p_{\texttt{guess}}(x|v_0).$$

Furthermore, if $x = 0$ *but* $\mathbf{b} = \bot$*, then a dishonest player has broadcast an incorrect version of a value to which he is committed by means of a linear combination of the commitments produced in the preprocessing phase.*

Proof

- Correctness: trivially, `ZeroTest` will open $[r \cdot 0] = [0]$.
- Soundness: by definition of the protocol, the output of $\texttt{ZeroTest}([x])$ is equal to $\top$ if and only if $\mathbf{b} = 0$, where

$$\mathbf{b} := (r - \tilde{r})(x - \tilde{x}) - \tilde{y}$$

where r is a variable uniformly distributed and independent of $x, \tilde{x}, \tilde{r}$ and $\tilde{y}$, and the variables $\tilde{r}, \tilde{x}$ and $\tilde{y}$ are chosen by the adversary, and are thus determined by his current view (since we can assume without loss of generality that the adversary is deterministic).

Now notice that for any $\hat{v}_0$ we have the following inequality:

$$p((r - \tilde{r}(\hat{v}_0))(x - \tilde{x}(\hat{v}_0)) = 0 | v_0 = \hat{v}_0) \leq 1/q + \max_{\hat{v}} p(x = \hat{x} | v_0 = \hat{v}_0)$$

In turn, by applying the law of total probability to $p((r - \tilde{r})(x - \tilde{x}) - \tilde{y} = 0)$ with the events $(v_0 = \hat{v}_0)$, we obtain the following inequality:

$$p((r - \tilde{r})(x - \tilde{x}) - \tilde{y} = 0) \leq 1/q + p_{\texttt{guess}}(x|v_0)\,.$$

Finally, if $x = 0$ but $\mathbf{b} = \bot$, then necessarily a player has communicated some incorrect values during `ZeroTest`; hence since all communications are performed by broadcast, he is committed to the incorrect value, so that the claim is proved. □

Finally, we need to discuss the privacy of `ZeroTest`; we first remark that Definition 1, formalizing our privacy notion, yields the following consequences:

Remark 1. Assume that the *uniform* distribution x is a list of size m given v; we then have the following properties:

(i) $p(x \in \ell) \leq m/q$ (immediate consequence of (I));
(ii) $p(x = y | x \notin \ell) \leq 1/(q-m)$ for any $y = y(v)$ (consequence of (II) via the law of total probability).

Furthermore, let r be a random variable independent of both v and x, and set $v' := (v, r)$. Then it trivially holds that if $p(x, v)$ satisfies the above definition, then so does $p(x, v')$.

Lemma 5. *Given a distribution $p(x, v_0)$, where v_0 denotes the adversary's view, assume that the* uniform *distribution of x given v_0 is a list of size m_0. Then after the execution of `ZeroTest`$([x])$, the distribution of x given v is a list of guesses of size at most $m := m_0 + 1$, where v denotes the adversary's view after the execution of `ZeroTest`.*

Proof. By looking at the instructions to compute and open $[xr]$ to P_i, we see that what the adversary can learn the following values (plus random sharings of them): $\gamma := x - a$, $\delta := r - b$ and $\pi := (r - \tilde{r})(x - \tilde{x})$, where a, b and r are jointly uniformly distributed and independent of each other and of $v, x, \tilde{x}, \tilde{r}$.

$\tilde{x}$ and $\tilde{r}$ are chosen by the adversary, and are thus determined by his view (since we assume without loss of generality that the adversary is deterministic).

Now given the adversary's view v_0 before the execution of `ZeroTest`, the adversary's *current* view is equal to $(v_0, \gamma, \delta, \pi)$; notice that a and b are (jointly) random and independent of x, r, v_0 and π, and thus so are $\gamma = x - a$ and $\delta = r - b$, so that we may restrict the view to $v := (v_0, \pi)$ (cf. Remark 1)[10].

Now by inductive hypothesis, there exists a conditional distribution $p(\ell_0 | v_0)$ such that properties I and II hold for $p(x, v_0, \ell_0) := p(x, v_0) \cdot p(\ell_0 | v_0)$; in a natural way, we define the new distribution to be

$$p(\ell = (x_1, \cdots, x_{m_0}, x_m) | v) := p(\ell_0 = (x_1, \cdots, x_{m_0}) | v_0) \cdot p(\tilde{x}(v_0) = x_m | v_0).$$

We now prove that properties I and II hold for $p(x, v, \ell)$: first of all, notice that $p\,(x \in \ell) = p\,(x \in \ell_0) + p\,(x = \tilde{x}(v_0) | x \notin \ell_0) \cdot p\,(x \notin \ell_0)$. Hence thanks to Remark 1 we have that

$$p\,(x \in \ell) \leq m_0/q + \big(1/(q - m_0)\big) \cdot \big((q - m_0)/q\big) = m/q\,.$$

Hence property I holds; we can thus focus on property II. As a first step, notice that

$$p(x | v = (\hat{v}_0, \hat{\pi}), \ell = \hat{\ell}, x \notin \hat{\ell}) = p(x | v_0 = \hat{v}_0, \ell_0 = \hat{\ell}_0, x \notin \hat{\ell}_0, x \neq \tilde{x}(\hat{v}_0))$$

[10] For the same reason, we omit here the fact that the view also contain random sharings of $x - a$, $r - b$ and π.

since if $x \notin \hat{\ell}$, then in particular $x \neq \tilde{x}(\hat{v}_0)$; hence we can re-write $\pi = \hat{\pi}$ as $r = \tilde{r}(v_0) + \hat{\pi}/(x - \tilde{x}(\hat{v}_0))$, and can be removed because r is independent of x, v_0 and ℓ_0. We thus get the following equality:

$$p(x|(v_0, \pi) = (\hat{v}_0, \hat{\pi}), \ell = \hat{\ell}, x \notin \hat{\ell}) = p(x|x \notin \hat{\ell}_0, x \neq \tilde{x}(\hat{v}_0))$$

which means that property II holds. □

A.2 The Complexity of the Block Check

We briefly discuss in this section the complexity of `BlockCheck`, which was presented in Sect. 3.4. First notice that since each block contains at most $|C|/n$ gates, there are at most $2|C|/n$ multiplication opening values to be checked in each block; we thus get the following complexity:

- $4n$ commitments need to be prepared, broadcast and opened (n to produce a random seed via `Rand`, and $3n$ during `ZeroTest`);
- the computational complexity of a block check is in $O\left(|C| + n^2\right)$ field operations (excluding computation on commitments), essentially given by the cost of computing the linear combination of the values to be checked;
- finally, the block check requires broadcasting $3n$ field elements for the dispute phase of `PublicOpening`. Notice that we do not use point-to-point communication.

B The Commitment Check

We now discuss how to authenticate shares of a value; as remarked in Sect. 2, for every shared value z that is $[\cdot]$-shared in the pre-processing phase each player P_i holds randomness ρ_{z_i} and the value $e_{z_i} := \texttt{Enc}(z_i, \rho_{z_i})$ has been broadcast. We give here the details on how to use these encryptions as a commitment scheme:

`EncryptionCheck`:
the protocol takes as input the index i of a player P_i and his share $z_i = \sum_{h=1}^{M} \lambda^{(h)} z_i^{(h)}$, where all $\left[z^{(h)}\right]$ are computed in the preprocessing phase; let $e_i^{(h)} := \texttt{Enc}\left(z_i^{(h)}, \rho_i^{(h)}\right)$ (these values are public, cf. Section 2).

(i) Players set $e_i := \sum_{h=1}^{M} \lambda^{(h)} e_i^{(h)}$;
(ii) P_i computes and broadcasts $\rho_i := \sum_{h=1}^{M} \lambda^{(h)} \rho_i^{(h)}$;
(iii) players set $e_i \leftarrow \texttt{Enc}\,(z_i, \rho_i)$.

If $e_i = \texttt{Enc}\,(z_i, \rho_i)$, the protocol outputs $\top$; otherwise, it outputs $\bot$.

Trivially, if P_i behaves honestly, `EncryptionCheck` will output $\top$; on the other hand, if the share $\tilde{z}_i$ he submitted is not correct, then the output will be $\bot$ since $\texttt{Enc}\,(z_i, \rho_i) \neq \texttt{Enc}\,(\tilde{z}_i, \tilde{\rho}_i)$ for any possible randomness $\tilde{\rho}_i$.

We are now ready to define the protocol `CommitCheck`, that simply applies `EncryptionCheck` to all shares submitted during the multiplication of two values:

`CommitCheck`:
The protocol takes as input the index i of a player P_i, a shared value $\langle z \rangle$ and the values $(\langle a \rangle, \langle b \rangle, \langle c \rangle), \langle r \rangle$ used to check its tag.

(i) Run `EncryptionCheck`(z_i) for P_i, denote by $\mathbf{b}_1$ its output;
(ii) run `EncryptionCheck`$(\gamma(z)_i - \tilde{z}\alpha_i - a_i)$ for P_i, denote by $\mathbf{b}_2$ its output;
(iii) run `EncryptionCheck`$(r_i - b_i)$ for P_i, denote by $\mathbf{b}_3$ its output;
(iv) run `EncryptionCheck`$(c_i + (\gamma(z) - \tilde{z}\alpha - a)\, b_i + (r - b)a_i + (\gamma(z) - \tilde{z}\alpha - a)(r - b))$ for P_i, denote by $\mathbf{b}_4$ its output.

Output $\mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \mathbf{b}_3 \wedge \mathbf{b}_4$.

The following proposition summarizes the security property of `CommitCheck`; we omit the proof, as it can be easily deduced from the definition of the protocol.

Proposition 7. *Under the binding property of the underlying commitment scheme, if a dishonest player has broadcast as part of* `BlockCheck` *an incorrect value, then this player will be publicly identified by* `CommitCheck`*. Furthermore, no honest player will incorrectly be identified as being dishonest.*

C Checking the Input and Output of the Computation

We show in this section how to secure the input-sharing and output-reconstruction phases; we use the main ideas and techniques of the multiplication check.

We first describe in more detail how the input sharing is performed in the original SPDZ protocol: each shared value $\langle r \rangle$ produce in the preprocessing phase comes with another type of sharing, denoted by

$$[\![r]\!] := \left([r], \left(\beta_i, \gamma(r)_1^i, \cdots, \gamma(r)_n^i\right)_{i=1,\cdots,n}\right),$$

where each player P_i holds $r_i, \beta_i, \gamma(r)_1^i, \cdots, \gamma(r)_n^i$ and $r\beta_i = \sum_j \gamma(r)_i^j$ for any i. Now in classical SPDZ, whenever a player P_i holds input x, a random shared value $[\![r]\!]$ is selected; then each player P_j communicates r_j and $\gamma(r)_i^j$ to P_i, who computes r and checks that $r\beta_i = \sum_j \gamma(r)_i^j$; P_i can then broadcast either an error message or the value $\varepsilon := x - r$. The input is then shared as $\langle r \rangle + \varepsilon$.

We add to this protocol our system of accusations and, as a last resort, the commitment checks:

`InputShare`:
The protocol is used to share an input x held by player P_i; a fresh king player P_k and a shared value $\langle r \rangle, [\![r]\!]$ are selected.

(i) for each $j \neq i$, player P_j sends $(r_j, \gamma(r)_i^j)$ to P_k, who in turn communicates these elements to P_i.
(ii) P_i computes $y := r\beta_i - \sum_j \gamma(r)_i^j$. If $y = 0$, he broadcasts $(\top, \varepsilon := x - r)$; players share x as $\langle r \rangle + \varepsilon$ and the protocol stops.
(iii) If $y \neq 0$, P_i broadcasts $\perp$. Then for each $j \neq i$, player P_j broadcasts $(r_j, \gamma(r)_i^j)$.
(iv) The king player P_k broadcast a list L of players that he accuses of inconsistent behaviour; if $L \neq \emptyset$, then P_k and all the players in L are added to the list of suspect players. `InputShare` is then rebooted; if all player are suspect, then the overall protocol aborts.
(v) If $L = \emptyset$, then P_i can accuse P_k of inconsistent behaviour; if that is the case, P_i and P_k are added to the list of suspect players, and the protocol is rebooted. If all player are suspect, then the overall protocol aborts.
(vi) Given that all values are now public, players run $\texttt{EncryptionCheck}(r_j)$ and $\texttt{EncryptionCheck}\left(\gamma(r)_i^j\right)$. If all players pass the encryption check, P_i is deemed dishonest and the protocol aborts

The following proposition follows from the definition of `InputShare` and proves that the protocol is secure:

Proposition 8. *Let x be an input held by player P_i; `InputShare` satisfies the following properties:*

- Correctness: *if players behave honestly, $\texttt{InputShare}(x)$ produces no accusations and players obtain a $\langle \cdot \rangle$-sharing of x.*
- Soundness: *if a player different from P_i behaves dishonestly during the execution of `InputShare`, then except with probability $1/q$ he will be deemed suspect or dishonest.*
- Privacy: *if P_i is honest, the adversary's guessing probability of x is equal to* $\max_{\hat{x}} p(x = \hat{x})$.

We now introduce an output-checking phase which makes use of the protocols introduced in the previous sections: it simply reconstructs the output, then checks its tag with `ZeroTest` and, if an error is detected, requires player to authenticate their shares via `CommitCheck`.

`OutputCheck`:
The protocol takes as input the shared value $\langle z \rangle$, output of the circuit.

(i) Each player P_i broadcasts his share z_i of $[z]$;

(ii) players set $\tilde{z} \leftarrow \sum_i z_i$; they then run `ZeroTest` $([\gamma(z)] - \tilde{z}[\alpha])$. Denote by $\big(\mathbf{b}, (\langle a\rangle, \langle b\rangle, \langle c\rangle), \langle r\rangle\big)$ its output;
(iii) if $\mathbf{b} = \top$, the protocol stops and output outputs $\tilde{z}$;
(iv) if $\mathbf{b} = \bot$, then player run $\mathbf{b}(i) \leftarrow$ `CommitCheck`$(\langle z\rangle)$ with values $(\langle a\rangle, \langle b\rangle, \langle c\rangle), \langle r\rangle$ for each player P_i; if $\mathbf{b}(i) = \bot$, the protocols outputs the message "P_i dishonest" and stops.

The following proposition proves that the protocol is correct and sound; we omit the proof here, as it can be easily obtained from the definition of `OutputCheck`, and refer to the full version of the paper for the details.

Proposition 9. *`OutputCheck` satisfies the following properties:*

- Correctness: *if players submit the correct shares of* $[z]$ *and behave honestly during `ZeroTest`, then `OutputCheck` will output the correct value* z*;*
- Security: *assume that* $\tilde{z} \neq z$ *or that the adversary behaved dishonestly in the `ZeroTest` phase; then `OutputCheck` will produce an accusation to a dishonest player except with probability* $1/q + p_v$*, where* p_v *is the adversary's guessing probability of* α *given his view* v*.*
  In the concrete setting, this error probability will be equal to $1/q + 1/(q - 2n)$*.*

References

1. Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. IACR Cryptology ePrint Archive 2016:187 (2016)
2. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC 1988, pp. 1–10. ACM (1988)
3. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20465-4_11
4. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS 2001, pp. 136–145. IEEE Computer Society (2001)
5. Chaum, D., Crépeau, C., Damgard, I.: Multiparty unconditionally secure protocols. In: STOC 1988, pp. 11–19. ACM (1988)
6. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40203-6_1
7. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_38

8. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC 1987, pp. 218–229. ACM (1987)
9. Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 369–386. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_21
10. Yao, A.C.: Protocols for secure computations. In: FOCS, pp. 160–164. IEEE Computer Society (1982)