Turbospeedz: Double Your Online SPDZ!
Improving SPDZ using Function Dependent Preprocessing

Aner Ben-Efraim* and Eran Omri*

Department of Computer Science, Ariel University, Ariel, Israel
anermosh@post.bgu.ac.il
omrier@gmail.com

Abstract. Secure multiparty computation allows a set of mutually distrusting parties to securely compute a function of their private inputs, revealing only the output, even if some of the parties are corrupt. Recent years have seen an enormous amount of work that drastically improved the concrete efficiency of secure multiparty computation protocols. Many secure multiparty protocols work in an "offline-online" model. In this model, the computation is split into two main phases: a relatively slow "offline phase", which the parties execute before they know their input, and a fast "online phase", which the parties execute after receiving their input.
One of the most popular and efficient protocols for secure multiparty computation working in this model is the SPDZ protocol (Damgård et al., CRYPTO 2012). The SPDZ offline phase is function independent, i.e., does not requires knowledge of the computed function at the offline phase. Thus, a natural question is: can the efficiency of the SPDZ protocol be improved if the function is known at the offline phase?
In this work, we answer the above question affirmatively. We show that by using a function dependent preprocessing protocol, the online communication of the SPDZ protocol can be brought down significantly, almost by a factor of 2, and the online computation is often also significantly reduced. In scenarios where communication is the bottleneck, such as strong computers on low bandwidth networks, this could potentially almost double the online throughput of the SPDZ protocol, when securely computing the same circuit many times in parallel (on different inputs).
We present two versions of our protocol: Our first version uses the SPDZ offline phase protocol as a black-box, which achieves the improved online communication at the cost of slightly increasing the offline communication. Our second version works by modifying the state-of-the-art SPDZ preprocessing protocol, Overdrive (Keller et al., Eurocrypt 2018). This version improves the overall communication over the state-of-the-art SPDZ when the function is known at the offline phase.

Keywords: Secure Multiparty Computation, SPDZ, concrete efficiency, offline/online

1 Introduction

Secure multiparty computation allows a set of mutually distrusting parties to securely compute a function of their private inputs, revealing only the output, even if some of the parties are corrupt. Secure computation was introduced by Yao [39] for 2 parties and by Goldreich et al. [24] for the multiparty case. Soon afterwards strong feasibility results were established, e.g., [24, 12, 9, 14, 36, 32]. Establishment of feasibility led to research of efficiency, and a long series of works, [28, 17, 18, 27, 20, 1, 4] and others, reduced the asymptotic communication and computational complexity of secure computation to almost optimal.

However, many of these asymptotically efficient protocols perform poorly when it comes to real world applications. For example, secure multiparty protocols based on fully-homomorphic encryption, e.g., [4], have almost optimal communication and computational complexity, but their concrete efficiency (i.e., their run-time in practice on real world problems) makes them somewhat impractical. The necessity of secure computation in real-world applications has therefore encouraged the study of concretely efficient protocols.

*Research supported by ISF grant 152/17.

Recent years have seen an enormous amount of work in this direction, and the concrete efficiency of secure multiparty computation protocols has also been significantly improved, e.g., [15, 22, 21, 33, 11, 38].

Real world scenarios have also led to the study of the "preprocessing" or "offline-online" model. In this model, the parties run a relatively expensive "offline" phase, i.e., a preprocessing protocol, before they know their inputs. After receiving their inputs, the parties then run a very efficient "online" protocol that uses the computations made at the offline phase. Such protocols can be used to make "real-time" secure computations, when it is known well in advance that these computations would take place.

Some of the works in the offline-online model, most notably SPDZ [22, 21], have a function independent offline phase, i.e., they assume the parties do not know the function to be computed during the preprocessing. Other works, such as most concretely efficient constant round secure multiparty protocols [34, 11, 26, 38], have a function dependent offline phase, i.e., they assume the parties already know the function at the expensive offline phase.

Both function independent offline and function dependent offline make sense in different real world applications. For example, assume a set of parties wish to securely compute an online auction at a specific date. On the one hand, a function dependent offline phase might allow the parties to run the auction more quickly. On the other hand, a function independent preprocessing would allow the parties more flexibility, such as changing the auction details up to the last minute. Therefore, it is important to study both of these models. In fact, several recent works in concretely efficient secure multiparty protocols, e.g., [34, 26, 38], separate the offline phase into two parts: a function independent preprocessing phase and a function dependent preprocessing phase.

Despite the SPDZ protocol being one of the most popular and efficient secure multiparty computation protocols, we observe that all previous works on the SPDZ protocol, e.g. [22, 21, 16, 6, 37], have only a function independent preprocessing phase. Thus, a natural question arises:

Question 1. Can the famous SPDZ protocol profit from using a function dependent preprocessing protocol?

1.1 Our Contribution and Techniques

In this work we answer Question 1 affirmatively. We present a new "SPDZ-like" protocol for secure computation against malicious adversaries, which requires approximately only half the communication in the online phase compared to the current state-of-the-art SPDZ online [21]. Furthermore, our protocol requires approximately only half the computation in the online MACCheck protocol, which is one of the main computational costs of the SPDZ online phase. Thus, we expect that our protocol could almost double the online throughput compared to SPDZ in some scenarios, e.g., strong machines on low-bandwidth networks that securely compute the same function multiple times in parallel on different inputs. We remark that these scenarios are of interest in real world applications; for example, increasing the throughput of secure multiple parallel AES computations has been discussed in [2] for Kerberos.

Our technique can be seen as adding additional randomness to the SPDZ protocol, so that more values can be seen in the clear. These values, which appear random to the adversary, aid in computing the secret-shared values of the following gates. More specifically, in our preprocessing protocol, the parties compute at each output wire of a multiplication gate an additional secret-shared random value. At the online phase, the parties reveal the sum of the real value and this random value. Then, using this revealed sum and additional information revealed at our preprocessing protocol, the parties can *locally* compute the shares for the output wires of the multiplication gates of the following layer in the circuit. Therefore, the only communication in the online phase for multiplication gates is revealing a single secret-shared value – the sum of the real value and a random value – whereas in [21] they reveal two values at each multiplication gate. However, our preprocessing protocol requires knowledge of the computed function. So in contrast to previous works on SPDZ, such as [22, 21, 16, 6, 37], our protocol's offline phase is function dependent.

For intuition, one could see this as follows: the extra randomness allows "shifting" the revealed values from corresponding to the input wires of multiplication gates (2 input wires for each gate) to corresponding to the output wires of the multiplication gates (1 output wire per gate) in the previous layer of the circuit. This "shift" is made possible because we know the function at the offline phase. Thus, our online protocol requires only half the amount of revealed values per multiplication gate compared to the improved online SPDZ [21], and additionally, we also save revealing values for input wires by observing that random values used for input distribution, which are "thrown out" in SPDZ, can be reused securely in our protocol. Since almost all the online communication comes from the revealing of these values, our online protocol requires almost half the amount of communication required by the online protocol in [21].[1] We further observe that the number of revealed values directly affects the amount of computation in the SPDZ MACCheck protocol, which is run at the end of the online phase to verify no cheating has occurred.

We show two variants to achieve our improved online phase. In the first version, we build on top of the SPDZ offline phase; that is, we first run the SPDZ offline preprocessing protocol (with values as detailed below), and then run another preprocessing protocol. Our additional preprocessing protocol is constant-round and its communication and computation is comparable to the SPDZ online phase, i.e., relatively small compared to the main cost of the SPDZ offline phase.

The number of Beaver triples (see Section 2 for the definition of Beaver triples) we need for the our protocol is exactly the same as in SPDZ. In this version of our protocol, we also require generating an additional (in comparison with SPDZ) random shared value for each multiplication gate at the function independent preprocessing. However, we note that generating a random shared value is significantly cheaper than generating a Beaver triple. Thus, in total, our offline phase, including both our new function dependent preprocessing protocol and the SPDZ preprocessing with additional random shared values, should not be significantly worse than the SPDZ offline phase.

Our second manner for achieving our improved online is by modifying the state-of-the-art SPDZ preprocessing protocol, Overdrive [31]. In this version of our protocol, we "align" randomness generated in Overdrive with the randomness needed for our online protocol. As a result, in this second version of the protocol we require at most the same amount of offline communication as Overdrive, and in some cases (depending on the computed circuit) even less. In order to "align" the randomness, our offline phase requires some extra computation (compared to Overdrive), but this computation consists of simple additions. Experiments in Overdrive suggest that communication is often the bottleneck also in the offline phase. Therefore, we expect our offline to improve the offline time in many instances (since we save communication), and even in circuits where we do not save offline communication, the extra additions needed for our protocol should not significantly increase the offline time.

To summarize, our protocol significantly improves over the SPDZ protocol in the online phase. If we modify the state-of-the-art Overdrive, we also improve the overall communication, while if we use the SPDZ offline phase as a black-box our offline phase is only slightly worse (which is still desirable in accordance with the spirit of the offline-online model). Thus, in cases where the computed function is known in advance, our protocol should be preferred over SPDZ.

1.2 Related Works

Works that focus on the preprocessing phase of the SPDZ protocol, such as MASCOT [29], Overdrive [31], and others, e.g., [7], are somewhat complementary to our work, since we use them for our offline phase. In the first version of our protocol we use these protocols as a black-box. In the second version of our protocol, we show that if the function is known at the offline phase, Overdrive can be slightly modified to "align" with our online protocol, increasing the efficiency (of the overall time) even further.[2]

There have also been several works that modified the SPDZ online phase in order to achieve additional properties, such as public auditability [6], efficient cheater detection [37], and extension to the integers modulo 2^k [16]. It is interesting to check if our ideas can also be used to improve the online phase in these protocols.

[1] Additional online communication includes squaring gates and communication in the MACCheck protocol, where we do not improve over [21]. However, this communication is relatively small, especially in large circuits. Therefore, our online communication is only slightly more than half the online communication of [21].

[2] We note that our "aligning" method works even better with the SPDZ preprocessing of [21], but the overall improvement would still probably not surpass using Overdrive. In contrast, due to a randomization technique used in MASCOT [29] triple generation, it is not clear if this "alignment" can also be applied to MASCOT preprocessing.

Another line of related works is secure computation based on lookup-tables, e.g. [23, 30]. Similarly to our work, these protocols require a function dependent offline phase, and further require sending only a single field element to each other party per gate at the online phase. Furthermore, protocols based on lookup-tables can have significantly more sophisticated gates, whereas we only have addition and multiplication gates. However, protocols based on lookup-tables require memory that grows linearly with the size of the field, per gate. Therefore, in contrast to our protocol (and SPDZ), protocols based on lookup-tables are useful mainly over small fields.[3]

A different approach to secure multiparty computation is based on garbled circuits. In the garbled circuit approach, the parties in some sense encrypt the function circuit at the offline phase. Then, at the online phase, the parties reveal keys for the inputs and locally compute the output of the circuit. This approach, for the multiparty setting, was originally proposed by Beaver et al. [9], and has recently received significant attention for concrete efficiency in several works, e.g., [34, 11, 26, 38, 10]. In contrast to SPDZ and protocols based on look-up tables, these protocols are constant round. Thus, the main advantage of secure multiparty protocols based on garbled circuits is to reduce the online time of deep circuits over high-latency networks. However, due to their large online computation complexity (for a large number of parties) they are generally not suitable for a high-throughput online goal, which is the main advantage of our protocol. Furthermore, it was shown (e.g., [11]) that in low-latency networks (e.g., LAN), protocols based on garbled circuits perform relatively poorly. And last, similarly to protocols based on lookup-tables, protocols based on garbled circuits require memory that is linear in the field size per gate, and are therefore impractical over large fields.[3]

We remark that there are also secure computation protocols which are specialized for restricted scenarios such as a semi-honest adversary, an honest majority, and/or a small number of parties, e.g. [19, 2, 3, 25]. We cannot compete with these protocols since achieving malicious security for any number of corrupt parties is significantly harder.

Regarding technique, our method can be seen as optimizing the computation by computing the gates on random values revealed on the wires. Similar ideas have been considered in previous works in various scenarios, such as the point-and-permute technique for garbled circuits [9] (that this technique implies computing the gates on revealed random values is seen more clearly in arithmetic garbled circuits [35, 5, 10]), in protocols based on look-up tables [23, 30], in protocols for an honest majority, e.g., [19], and recently for an extremely efficient protocol for 4 parties with an honest majority [25]. Our protocols show that this technique can also be used to improve the well studied SPDZ protocol.

Organization. In Section 2 we recall the ideas of the SPDZ protocol and Overdrive. In Section 3 we describe our function dependent offline protocols that uses SPDZ offline as a black-box, and our new online protocol. In Section 4 we explain how to improve the overall time of our protocol when using Overdrive. In Section 5 we prove correctness and the security of our protocols.

Notation and conventions. Similarly to SPDZ, throughout this paper we assume the computation is performed by n parties over some finite field $\mathbb{F}$. We also assume that $|\mathbb{F}|$ is exponential in the security parameter κ. When we refer to the computed function, we assume it is encoded as an arithmetic circuit C over $\mathbb{F}$.

2 Review of the SPDZ Protocol and Overdrive

In this section, we briefly review the (improved) SPDZ protocol presented in [21]. Then we partially explain how Overdrive [31] generates multiplication triples using a public-key semi-homomorphic encryption. We follow [21] for SPDZ because it has the currently most efficient online phase. Overdrive [31] is currently the state-of-the-art protocol for generating multiplication triples. The results in this section are given only as a preliminary to our work in the following sections, and are taken mainly from [21] and [31].

[3]To be more precise, these protocols perform best over small characteristic fields. However, they can be somewhat efficiently extended to arithmetic computations over the integers using the Chinese Remainder Theorem, e.g., [5, 10], and to extension fields with small characteristic using multiplication embedding.

The SPDZ protocol executes a relatively expensive "offline" preprocessing phase in order to achieve a very efficient online phase, which is secure against any number of corruptions (in the model of security with abort). Before giving an overview of the SPDZ protocol, we recall Beaver multiplication triples [8], which is one of the main building blocks of the SPDZ protocol.

Definitions of $[[\cdot]]$*-shared elements and Beaver multiplication triples.* Assume each party has a uniform additive share $\alpha_i \in \mathbb{F}$ of a secret global MAC value $\alpha = \Sigma_{i=1}^{n}\alpha_i$. An element $a \in \mathbb{F}$ is $[[\cdot]]$-shared if each party holds a pair $(a_i, \gamma(a)_i)$, where a_i is an additive secret-sharing of a, i.e., $a = \Sigma_{i=1}^{n} a_i$, and $\gamma(a)_i$ is an additive secret-sharing of $\gamma(a) = \alpha \cdot a$, i.e., $\gamma(\alpha) = \Sigma_{i=1}^{n}\gamma(a)_i$. For an element $a \in \mathbb{F}$ we denote $[[a]] \overset{\text{def}}{=} ((a_1, \dots, a_n), (\gamma(a)_1, \dots, \gamma(a)_n))$.

A nice feature of $[[\cdot]]$-shared elements is that addition of 2 $[[\cdot]]$-shared elements, addition of a public scalar, and multiplication by a public scalar can be computed locally.

Property 1. For $a, b, e \in \mathbb{F}$ with $[[a]], [[b]]$ being $[[\cdot]]$-shares of a,b respectively and e a public value

- $[[a]] + [[b]] \overset{\text{def}}{=} ((a_1 + b_1, \dots, a_n + b_n), (\gamma(a)_1 + \gamma(b)_1, \dots, \gamma(a)_n + \gamma(b)_n))$ is a $[[\cdot]]$-share of $a + b$,
- $e \cdot [[a]] \overset{\text{def}}{=} ((e \cdot a_1, \dots, e \cdot a_n), (e \cdot \gamma(a)_1, \dots, e \cdot \gamma(a)_n))$ is a $[[\cdot]]$-share of $e \cdot a$,
- $e + [[a]] \overset{\text{def}}{=} ((e + a_1, a_2, \dots, a_n), (\gamma(a)_1 + e \cdot \alpha_1, \dots, \gamma(a)_n + e \cdot \alpha_n))$ is a $[[\cdot]]$-share of $e + a$.

However, to perform multiplication of 2 $[[\cdot]]$-shared elements in the SPDZ protocol, the parties require interaction and a Beaver multiplication triple [8].

A Beaver multiplication triple is a triple, $([[a]], [[b]], [[c]])$, of $[[\cdot]]$-shared values such that $c = a \cdot b$. Similarly, a squaring pair is a pair, $([[a]], [[c]])$, of $[[\cdot]]$-shared values such that $c = a^2$; squaring pairs are used in [21] to compute the square of a $[[\cdot]]$-shared value more efficiently.

MACCheck protocol. During both the offline and the online phase of the SPDZ protocol, certain $[[\cdot]]$-shared values are (partially) revealed to some or all of the parties. I.e., the parties learn the shared value (but not the MAC). A malicious adversary may attempt to manipulate its shares to reveal different values than the ones actually shared. Thus, some procedure must be run to ensure such a cheating does not occur.

This procedure is the MACCheck protocol of [21], which receives a set of revealed $[[\cdot]]$-shared values and efficiently verifies, with failure probability $\leq \frac{2}{|\mathbb{F}|}$ ($|\mathbb{F}|$ being the size of the field), that no cheating has occurred. Note that in this paper we discuss only large fields (i.e., $\mathbb{F}$ is exponential in the security parameter), so MACCheck verifies that the adversary did not cheat except with negligible probability. For completeness, this protocol is given in Appendix C, but we shall only require the following claim:

Claim 1 (Informal) [21, Lemma 1] *Given a set of partially revealed* $[[\cdot]]$*-shared values, if the revealed values do not match the* $[[\cdot]]$*-shared values, MACCheck aborts except with probability* $\leq \frac{2}{|\mathbb{F}|}$*. Furthermore, if the adversary does not cheat, MACCheck leaks no information on the queried values, the global MAC* α*, and the honest parties' shares.*

The SPDZ offline phase. The main part of the SPDZ offline phase is a preprocessing protocol that securely generates Beaver triples and additional random $[[\cdot]]$-shared values. There have been several works that significantly improved the original SPDZ preprocessing protocol, e.g., [31, 29, 7]. The current state-of-the-art protocols are Overdrive [31] for large prime fields, which is based on semi-homomorphic encryption, and MASCOT [29] for large fields of characteristic 2, which is based on oblivious transfer.

In Section 3 we assume black-box access to the SPDZ offline functionality (which in practice would probably be implemented using Overdrive). I.e., we assume that the parties can access a functionality $\mathcal{F}_{\text{Prep}}$ that gives the parties the shares of the requested number of Beaver triples $([[a]], [[b]], [[c]])$, square pairs $([[a]], [[c]])$, random $[[\cdot]]$-shared elements $[[r]]$, and input maskings $(r_i, [[r_i]])$.[4] For completeness, the functionality $\mathcal{F}_{\text{Prep}}$, taken from [21], is given in Appendix D. For concrete protocols, one should look at [31, 29, 7, 21].

[4]An input masking $(r_i, [[r_i]])$ is a random $[[\cdot]]$-shared element, where the value r_i is known to party i.

In Section 4 we show how to modify Overdrive (SPDZ offline protocol) so that the values generated at the offline are "aligned" with the values needed in our online protocol. A partial overview of Overdrive, in particular of the triple generation protocol and the SPDZ sacrifice step, is given at the end of this section, and the modification is explained in Section 4.

The SPDZ online phase. As mentioned, one of the highlights of the SPDZ protocol is its very efficient online protocol that is secure against any number of corruptions (in the model of security with abort), which is achieved using the relatively expensive preprocessing protocol. In the online protocol, the parties first compute $[[\cdot]]$-shares of their inputs as follows: party i shares its input x_i by revealing $x_i - r_i$, where r_i is an input masking that was generated at the offline phase. The parties then locally compute $[[x_i]] \leftarrow [[r_i]] + (x_i - r_i)$ using Property 1.

Addition gates are computed locally: let the input wires be x, y and the output wire be z. The parties locally compute $[[z]] \leftarrow [[x]] + [[y]]$ using Property 1.

In order to compute multiplication gates, the parties use a Beaver triple $([[a]], [[b]], [[c]])$ as follows: let the input wires be x, y and the output wire be z. The parties locally compute $[[\epsilon]] \leftarrow [[x]] - [[a]]$ and $[[\rho]] \leftarrow [[y]] - [[b]]$, and then communicate to partially reveal ϵ and ρ. Then, the parties use Property 1 to locally compute

$$[[x \cdot y]] \leftarrow [[c]] + \epsilon \cdot [[b]] + \rho \cdot [[a]] + \epsilon \cdot \rho. \tag{1}$$

Squaring gates are computed in a similar but slightly simpler way, using a square pair.

At the end of the protocol, before outputting the result, the parties run a MACCheck protocol to verify that the corrupt parties did not cheat. If the corrupt parties did attempt to cheat, the cheating is detected with overwhelming probability, and the honest parties abort. Note that fairness is not guaranteed, i.e., the adversary can learn the output while the honest parties do not.

Triple Generation using Overdrive. Overdrive [31] (and previously [13]) construct multiplication triples using a public-key semi-homomorphic encryption Enc. For efficiency, Overdrive uses the BGV encryption that introduces noise, which needs to be "drowned" for security reasons. Furthermore, the parties need to prove that some encryptions are generated correctly using zero-knowledge proofs. Due to space constraints we do not go into the details here, and encourage the reader to read [31] for the details.

The multiplication scheme is as follows: assume the parties hold additive shares $a = \Sigma_i a_i, b = \Sigma_i b_i$, then $ab = (\Sigma_i a_i \cdot \Sigma b_i) = \Sigma_i a_i b_i + \Sigma_{i \neq j} a_i b_j$. Each $a_i b_i$ can be computed locally by party i, and (shares of) $a_i b_j$ are computed using the following two party protocol: Party i sends $\mathrm{Enc}(a_i)$ encrypted under its own public key. Party j, using Party i's public key and the received $\mathrm{Enc}(a_i)$, responds with $C_i = b_j \cdot \mathrm{Enc}(a_i) - \mathrm{Enc}(c_j)$, where c_j is a randomly chosen share. Then party i decrypts $c_i = \mathrm{Dec}(C_i)$ and by the homomorphic property $c_i + c_j = a_i b_j$, so (c_i, c_j) is a secret-sharing of $a_i b_j$.

The above multiplication is used in two places in Overdrive: **(1)** To compute the shares of $c = ab$ in the multiplication triple, and **(2)** To generate the MACed shares $[[a]], [[b]], [[c]]$; we shall assume the latter is done by calling the functionality $\mathcal{F}_{[[]]}$. An implementation of $\mathcal{F}_{[[]]}$ and the original Overdrive triple generation protocol can be found in [31, Figures 4 and 7]. Our modified version of the triple generation protocol, used for our protocol in Section 4, is given in Figure 3.

One issue that arises is that the adversary might attempt to cheat in the triple generation. For efficiency reasons, only some of this is captured in Overdrive using zero-knowledge proofs. In particular, the adversary is able to create triples $(a, b, ab + \mathsf{e})$ for some error e of her choice. This is solved in Overdrive using the "SPDZ sacrifice" – triples are generated in pairs, and one is "sacrificed" to ensure the other triple is correct. The SPDZ sacrifice was slightly improved in [29], showing that it suffices to use correlated triple pairs $([[a]], [[b]], [[c]]), ([[a]], [[\hat{b}]], [[\hat{c}]])$.

The (improved) SPDZ sacrifice works roughly as follows: a random element r is chosen *after* the triples are (possibly incorrectly) generated. Then, $\rho = rb - \hat{b}$ is partially opened. Using ρ, the parties compute (using Property 1) and partially reveal $\tau = rc - \hat{c} - \rho a$, and abort if $\tau \neq 0$. It can be shown that if the adversary cheated in generating the triples, i.e., $c = ab + \mathsf{e}$ and $\hat{c} = a\hat{b} + \hat{\mathsf{e}}$ with $e \neq 0$ and/or $\hat{e} \neq 0$, then $\tau \neq 0$ with overwhelming probability. If the adversary tries to cheat in the revealing of ρ and/or τ, she is later be caught by the MACCheck protocol with overwhelming probability. It is easy to see that if the adversary does not cheat then the parties do not abort. And because $\hat{b}$ and $\hat{c}$ are "sacrificed" (i.e., not used elsewhere in the protocol), ρ does not leak any information on $([[a]], [[b]], [[c]])$.

3 Our New Protocol, using SPDZ Offline as Black-Box

In this section we describe our two new protocols – our added function dependent offline protocol and our new "SPDZ-like" online protocol. The offline protocol in this section uses the SPDZ preprocessing protocol as a black-box. A more efficient version of our protocol achieved by modifying the state-of-the-art SPDZ preprocessing protocol, Overdrive, is given in Section 4.

We use slightly different notation and equations than the ones explained in Section 2, so we first give the details of our notation and equations.

3.1 Notation and Equations

Similarly to the online phase of the SPDZ protocol, in our online protocol the parties compute (in topological order on the circuit) for each wire a $[[\cdot]]$-shared value that corresponds to the real value on the wire. We denote the real value on wire ω by v_ω and correspondingly its $[[\cdot]]$-shared value by $[[v_\omega]]$.[5] Observe that the real values depend on the inputs, and are thus determined only at the online phase.

In our online protocol, the parties additionally hold at each wire ω shares of a random field element, which we term the *permutation element* and denote by λ_ω (and its $[[\cdot]]$-share by $[[\lambda_\omega]]$). The shares of these permutation elements are generated and computed in the offline phase. Note that these permutation elements are independent of the real values. It is also important that the permutation elements are independent of the multiplication triples, see Section B.

At the online phase, after computing the shares corresponding to the real value, the parties open the sum of the real value and the permutation element. We call this sum the *external value*, and denote it by $e_\omega \overset{\text{def}}{=} v_\omega + \lambda_\omega$. The important observation is that since the permutation element is independently random and unknown, the external value reveals no information on the real value; similar observations are implicitly used in SPDZ, e.g., when revealing ϵ, ρ of multiplication input wires.

For addition gates with input wires x, y, we let the permutation element of the output wire z be the sum of the permutation elements on the input wires, i.e., $\lambda_z = \lambda_x + \lambda_y$. Thus, the shares of λ_z can be computed locally by the parties from the shares of λ_x and λ_y. Furthermore, we observe that during the online phase, the external value on the output wire can also be computed locally by the parties, since the above implies that the external value of the output wire is the sum of the external values of the input wires:

$$e_z = v_z + \lambda_z = (v_x + v_y) + (\lambda_x + \lambda_y) = (v_x + \lambda_x) + (v_y + \lambda_y) = e_x + e_y \tag{2}$$

For a multiplication gate with input wires x and y and output wire z, assume the beaver triple $([[a]], [[b]], [[c]])$ is associated with the multiplication gate. We denote the *input offsets* by

$$\widetilde{\lambda_x} \overset{\text{def}}{=} a - \lambda_x \tag{3}$$

and

$$\widetilde{\lambda_y} \overset{\text{def}}{=} b - \lambda_y. \tag{4}$$

We further denote the *adjusted external values* on the input wires by

$$\widehat{e_x} \overset{\text{def}}{=} e_x + \widetilde{\lambda_x} = (v_x + \lambda_x) + (a - \lambda_x) = v_x + a, \tag{5}$$

[5]In [22, 21] they do not distinguish between the wire and its value – there v_ω and $[[v_\omega]]$ are denoted ω and $[[\omega]]$, respectively. Our notation is similar to notations used for multiparty garbled circuits, e.g., [9, 11].

$$\widehat{e_y} \overset{\text{def}}{=} e_y + \widetilde{\lambda_y} = (v_y + \lambda_y) + (b - \lambda_y) = v_y + b. \tag{6}$$

Then, we have the following resulting equation:

$$v_x v_y = (v_x + a)(v_y + b) - a(v_y + b) - b(v_x + a) + ab = \widehat{e}_x \widehat{e}_y - \widehat{e}_y a - \widehat{e}_x b + c \tag{7}$$

Equation (7) is used in our online protocol in order to compute the shares of the multiplication. For the output wire, we set the permutation element on the output wire to be $\lambda_z = c + r$, where r is a fresh random $[[\cdot]]$-shared value.[6]

Remark 1. Note that Equation (7) we use is slightly different than Equation (1) used in [22, 21] – Equation (1) uses the values $\epsilon = v_x - a$ and $\rho = v_y - b$ instead of the values $\widehat{e}_x = v_x + a$ and $\widehat{e}_y = v_y + b$. However, this change is only semantic.

Similarly to [21] we observe that squaring gates can be computed using a square pair, i.e., a pair $([[a]], [[c]])$ such that $c = a^2$, instead of a multiplication triple.Squaring gates are computed using the following equation:

$$(v_x)^2 = (v_x + a)^2 - 2a(v_y + a) + a^2 = (\widehat{e}_x)^2 - 2\widehat{e}_x a + c \tag{8}$$

Remark 2. In [21] squaring requires partially revealing only a single value, and therefore we do not have any saving over [21] for squaring gates. Due to the similarity with regular multiplication gates, we omit further discussion on squaring gates.

3.2 Function Dependent Offline Protocol

In this section we describe our new function dependent offline protocol and the functionality it implements. Our offline protocol and its functionality and simulator (in the proof, see Appendix 5.1) use the SPDZ offline protocol, functionality, and simulator. For completeness, we include the SPDZ offline functionality in Appendix D. For clarification, we denote the SPDZ offline of [21] using “Prep” and the new offline using “FDPrep”, i.e.,

– The protocols are denoted Π_{Prep} and Π_{FDPrep}.
– The functionalities are denoted $\mathcal{F}_{\text{Prep}}$ and $\mathcal{F}_{\text{FDPrep}}$.
– The simulators are denoted $\mathcal{S}_{\text{Prep}}$ and $\mathcal{S}_{\text{FDPrep}}$.

Our function dependent offline protocol is formally described in Figure 1. Our offline protocol runs the original SPDZ offline as a sub-protocol, and implements a very similar functionality to the SPDZ offline functionality. Our offline functionality is formally described in Figure 4. The main differences of our new offline from the original SPDZ offline are:

1. The new offline protocol/functionality receives the circuit as input. The original SPDZ protocol/functionality is then run with the number of multiplication gates and input wires as in the circuit. For each multiplication gate, the SPDZ offline also generates an additional random $[[\cdot]]$-shared element.
2. Each generated multiplication triple is associated with a specific multiplication gate. Similarly, each input wire is assigned a specific random $[[\cdot]]$-shared element revealed only to Party i.
3. For each multiplication gate, the protocol/functionality also associates a random $[[\cdot]]$-shared element, called the permutation element.
4. The protocol/functionality reveals specific “offset values”, where an “offset value” is the difference between 2 random $[[\cdot]]$-shared elements. The protocol runs a MACCheck on these revealed values to ensure the adversary did not cheat on any of these values.

Notice that all the offsets can be revealed in parallel. Therefore, we added only a constant number of communication rounds to the SPDZ preprocessing protocol.

[6]It might be tempting to naïvely set $\lambda_z = c$, but this would not work, as λ_z must be independently random, see Appendix B for details. However, in Section 4 we show that by modifying Overdrive this part can be optimized.

Protocol $\Pi_{\mathbf{FDPrep}}$

Initialize: The parties call Π_{Prep} (i.e., a secure implementation of $\mathcal{F}_{\text{Prep}}$) with the number of multiplication gates, squaring gates, and input wires to receive the desired number of input maskings, multiplication triples, squaring pairs, and random $[[\cdot]]$-shared elements.[a]
The parties then perform the following local computations in topological order on the gates of the circuit:
Input Wires: For every $i \in [n]$, for each input wire of party i the parties associate an available masking $(r_i, [[r_i]])$ (i.e., a random $[[\cdot]]$-shared element revealed to party i).
Addition gates: On an addition gate with input permutation element shares $[[\lambda_x]]$ and $[[\lambda_y]]$ the parties locally compute the output permutation element shares $[[\lambda_z]] \leftarrow [[\lambda_x]] + [[\lambda_y]]$.
Multiplication gates: On a multiplication gate with input permutation element shares $[[\lambda_x]]$ and $[[\lambda_y]]$ the parties assign to the gate the next available multiplication triple $([[a]], [[b]], [[c]])$ and the next available $[[\cdot]]$-shared random element $[[r]]$, and locally compute:

- $[[\cdot]]$-shares of the offset values $[[\widetilde{\lambda_x}]] \leftarrow [[a]] - [[\lambda_x]]$ and $[[\widetilde{\lambda_y}]] \leftarrow [[b]] - [[\lambda_y]]$.
- $[[\cdot]]$-shares of the permutation element on the output wire $[[\lambda_z]] \leftarrow [[c]] + [[r]]$.

Output: After performing all the above local computation, the parties partially reveal the offset values $\widetilde{\lambda_x}, \widetilde{\lambda_y}$ for every multiplication gate and $\widetilde{\lambda_x}$ for every squaring gate.
Output verification: This procedure is entered once the parties have finished the above function dependent preprocessing phase.
The parties call the MACCheck protocol with the input being all the opened values so far. If MACCheck fails, they output ϕ and abort, otherwise they accept the partially opened offset values.

[a] Recall that we require an additional $[[\cdot]]$-shared element for each multiplication/squaring gate.

Fig. 1: Our new function dependent preprocessing protocol

3.3 New Online Protocol

In this section we explain our new online protocol, which is formally given in Figure 2. As explained in the introduction, the main difference of our new online protocol from previous SPDZ online protocols, e.g., [22, 21], is that the parties have at each wire more values, which helps them compute the output values more efficiently. Concretely, in the SPDZ online phase only the real value on the wire is secret-shared amongst the parties. In our protocol, another random field element, the permutation element, is secret-shared amongst the parties. Furthermore, in our protocol the *external value*, i.e., the sum of the real value and the permutation element, is revealed to all the parties.

These external values, after certain adjustment, help in computing $[[\cdot]]$-shares of the output value of multiplication gates: In order to connect the shared and revealed values on the output wire to the input wires of the following multiplication gates, the parties use the revealed offsets from the function dependent preprocessing, to compute the adjusted external values on the input wires. The permutation elements that correspond to these adjusted external values match the shared values of the multiplication triples, which allows the parties to use Equation (7). Thus, the $[[\cdot]]$-shares of the product and the $[[\cdot]]$-shares of the output external value are in fact computed *locally*, and all that remains (to continue this process to the following gates) is to partially reveal the output external value.

We observe that we also save communication on input wires compared to [21] because we "reuse" the shares used for distribution of the input (r_i in **Input** part of Figure 2) by letting r_i be equal to the permutation element on that wire.

The main advantage of our new protocol over the SPDZ protocol of [21] is that it requires opening only a single value for each multiplication gate at the online phase. However, notice that in the function dependent preprocessing we open 2 additional values, so in total we open 1.5 times more values than SPDZ. To counter this undesirable side-effect, we present in Section 4 a more efficient version of our protocol that works by modifying Overdrive.

Protocol $\mathbf{\Pi}_{\text{Online}}$

Initialize: The parties call $\mathcal{F}_{\text{FDPrep}}$ (Figure 4 in Appendix 5.1) with the circuit to receive the masking values at each of the input wires and the random elements, the Beaver triples/squaring pairs, and the revealed offsets at each of the multiplication/squaring gates.
Input: To share his input v_{x_i}, party i takes the associated mask value $(r_i, [[r_i]])$ and does the following:

- Broadcast $e_{x_i} = v_{x_i} + r_i$.
- The parties compute $[[v_{x_i}]] \leftarrow e_{x_i} - [[r_i]]$.
- The parties store e_{x_i} as the external value on the wire, and $[[r_i]]$ as shares of the permutation element, λ_{x_i}.

Add: On an addition gate with input wires x, y, input shared values $([[v_x]], [[v_y]])$, input shared permutation elements $([[\lambda_x]], [[\lambda_y]])$, and input external values (e_x, e_y), the parties locally compute the following for the output wire z:

1. $[[v_z]] \leftarrow [[v_x]] + [[v_y]]$
2. $[[\lambda_z]] \leftarrow [[\lambda_x]] + [[\lambda_y]]$[a]
3. $e_z \leftarrow e_x + e_y$

Multiply: On a multiplication gate with input shared values $([[v_x]], [[v_y]])$and input external values (e_x, e_y), the parties take the associated multiplication triple $([[a]], [[b]], [[c]])$, the partially opened offsets $\widetilde{\lambda_x}, \widetilde{\lambda_y}$, and the associated random $[[\cdot]]$-shared value $[[r]]$ and perform the following steps:

1. Locally compute:
   - $\widehat{e_x} \leftarrow e_x + \widetilde{\lambda_x}$ and $\widehat{e_y} \leftarrow e_y + \widetilde{\lambda_y}$.
   - $[[v_z]] \leftarrow \widehat{e_x}\widehat{e_y} - \widehat{e_y}[[a]] - \widehat{e_x}[[b]] + [[c]]$.[b]
   - $[[\lambda_z]] \leftarrow [[c]] + [[r]]$.[a]
2. Partially open $[[e_z]] \leftarrow [[v_z]] + [[\lambda_z]]$.

Output: This procedure is entered once the parties have finished the circuit evaluation, but still the final output v_z has not been opened.

1. The parties call the MACCheck protocol with the input being all the opened values so far in the online phase. If MACCheck fails, they output ϕ and abort.[c]
2. The parties open v_z and call MACCheck with input v_z, to verify its MAC. If the check fails, they output ϕ and abort, otherwise they accept v_z as a valid output.

[a] This computation was performed at the function dependent offline phase.
[b] By Equation (7), v_z holds the value $v_x \cdot v_y$.
[c] ϕ represents that the corrupted parties remain undetected.

Fig. 2: Our new online phase protocol

4 Improvement via Modification in Overdrive

In this section we explain how to improve the overall time of our protocol, by modifying Overdrive (SPDZ offline protocol), instead of using it as a black-box. The main idea of this optimization is to **(1)** Avoid creating more random elements in the offline phase than in SPDZ, and **(2)** Avoid partially opening more elements in the offline phase than in SPDZ.

A first naïve attempt might be to set the permutation element of the output wire to equal c of the multiplication triple, but we show this is insecure in Appendix B. In contrast, if the output wire is an input to a multiplication gate, then setting the permutation element to equal a of the multiplication triple of the following gate is secure. It turns out that by a slight tweak, this can also be extended to general arithmetic circuits.

Furthermore, the efficiency can be even further improved by setting all a's corresponding to the same wire to be equal. Clearly, this cannot be done using SPDZ offline in a black-box fashion, since SPDZ offline generates independently random multiplication triples. Therefore, it is not clear that this optimization can be achieved for every SPDZ offline protocol. Nevertheless, we show that it is possible to achieve this optimization securely by modifying some SPDZ offline protocols, and in particular Overdrive, which is currently the state-of-the-art SPDZ preprocessing protocol. The formal details of creating these correlated triples are given in Figure 3.

Protocol $\Pi_{\mathbf{FDTriple}}$

Initialize: Each party P_i randomly chooses $\lambda_{\omega,i} \leftarrow \mathbb{F}$ for every wire ω that is an input wire of the circuit[a] or an output wire of a multiplication gate. Then, in topological order on the circuit, for each addition gate with input wires x, y and output wire z, party P_i computes $\lambda_{z,i} = \lambda_{x,i} + \lambda_{y,i}$.
Multiplication: For each multiplication gate with input wires x, y

1. Each party P_i sets $a_i = \lambda_{x,i}, b_i = \lambda_{y,i}$ and randomly selects $\hat{b}_i \leftarrow \mathbb{F}$.
2. Every unordered pair (P_i, P_j) executes the following 2-party multiplication as in [31]:
   (a) P_i sends P_j the encryption $\mathrm{Enc}_i(a_i)$.[b]
   (b) P_j responds with $C^{(ij)} = b_j \cdot \mathrm{Enc}_i(a_i) - \mathrm{Enc}_i(e^{(ij)})$ for a random $e^{(ij)} \leftarrow \mathbb{F}$.[b]
   (c) P_i decrypts $d^{(ij)} = \mathrm{Dec}_i(C^{(ij)})$.
   (d) The last two steps are repeated with $\hat{b}_j$ to get $\hat{e}^{(ij)}$ and $\hat{d}^{(ij)}$.
3. Each party P_i computes $c_i = a_i b_i + \Sigma_{i \neq j}(e^{(ji)} + d^{(ij)})$, and $\hat{c}_i$ similarly.

Authentication: Each party P_i inputs to $\mathcal{F}_{[[]]}$ the shares $\lambda_{\omega,i}$ for each wire ω that is an input wire of the circuit or an output wire of a multiplication gate, and additionally the shares $c_i, \hat{b}_i$, and $\hat{c}_i$ for each multiplication gate; for each such value the functionality outputs to the parties $[[\lambda_\omega]]$, $[[c]]$, $[[\hat{b}]]$, or $[[\hat{c}]]$, respectively, where $\lambda_\omega = \Sigma_i \lambda_{\omega,i}$, etc.
Sacrifice: The parties do the following for each multiplication triple pair $([[a]], [[b]], [[c]]), ([[a]], [[\hat{b}]], [[\hat{c}]])$:

1. Call $r \leftarrow \mathcal{F}_{\mathrm{RAND}}$.
2. Compute and partially open $[[\rho]] = r[[b]] - [[\hat{b}]]$.[c]
3. Compute and partially open $[[\tau]] = r \cdot [[c]] - [[\hat{c}]] - \rho \cdot [[a]]$.[c] If $\tau \neq 0$ then abort.

MACCheck and Output: Run MACCheck on all opened values. If the check fails then abort. Otherwise, output all non-sacrificed computed triples $([[a]], [[b]], [[c]])$.

[a]The value $\lambda_\omega = \Sigma_i \lambda_{\omega,i}$ is the random value used in Π_{Online} for the input distribution, and therefore is later partially revealed to the relevant party.

[b]For simplicity, the details of the zero-knowledge proofs and of the noise drowning have been omitted; the complete details of this 2-party protocol can be found in [31, Figure 7].

[c]Note that a and b are linear combinations of the permutation elements λ_ω input to $\mathcal{F}_{[[]]}$, and thus $[[a]]$ and $[[b]]$ can be locally computed by the parties.

Fig. 3: Our modified triple generation protocol

The result of this optimization is that $\widetilde{\lambda_x}, \widetilde{\lambda_y}$ in Equations (3),(4) always equal 0, implying that during the online phase the adjusted external values $\widehat{e}$ are equal to the external values e on the gates' input wires. Thus, this optimization also slightly simplifies and improves our online protocol.

Additionally, depending on the circuit, in some cases the same encryptions $\text{Enc}(a_i)$ in Step 2a could be used in several multiplications. For example, this may be possible when the same wire is input to several gates[7] (or even using the semi-homomorphic property $\text{Enc}(a + a') = \text{Enc}(a) + \text{Enc}(a')$). Reusing the same encryption reduces both the computation and communication, and choosing which wires should play a and b in the 2-party protocol to gain maximal reduction can be computed based solely on the circuit.

Since the only operations we perform in addition to those already necessary in Overdrive are additions corresponding to addition gates, we do not increase communication and only slightly increase computation in the worst case. Furthermore, due to reusing the encryptions, in many circuits our offline protocol will even have less computation and communication then using Overdrive for generating independent triples.

Note that since we changed the triple generation, it implies that in the SPDZ sacrifice step the shares of a and b in the multiplication step now correspond to a linear combination of shares input to $\mathcal{F}_{[[]]}$. Therefore, we must show that this change maintains the security. Recall that the security requirements from the SPDZ sacrifice are that **(1)** If the adversary cheats and sets $c = ab + \mathsf{e}$ or $\hat{c} = a\hat{b} + \hat{\mathsf{e}}$ with $e \neq 0$ and/or $\hat{\mathsf{e}} \neq 0$ then the honest parties abort with overwhelming probability, and **(2)** If $\hat{b}$ and $\hat{c}$ are "sacrificed" (not used later in the protocol) then no information is leaked on $([[a]], [[b]], [[c]])$.

The proof is similar to the proof of the original SPDZ sacrifice, and given in Appendix 5.2. Two crucial points are that **(1)** a and b are linear combinations of the permutation elements (which are input into $\mathcal{F}_{[[]]}$) and thus so are tb and ρa (when ρ is treated as a constant), and that **(2)** $\hat{b}$ is independently and randomly chosen for each sacrifice, and therefore $\rho = tb - \hat{b}$ revels nothing even if some of the b's in different multiplications are correlated (or even equal).

5 Correctness and Security

In this section we explain the correctness and security of our protocol.

Correctness. Assuming no party tries to cheat, the correctness follows from observing that at each wire

- The parties hold shares $[[v_\omega]]$ of the correct real value v_ω,
- The revealed external value e_ω corresponds to the sum of the real value v_ω and the (shared) permutation element λ_ω.

This statement is proved by induction in topological order on the wires:

- For input wires this follows from the Input part in Protocol Π_{online}.
- For output wires of addition gates, the claim on the real values follows from Property 1, and the claim on the external value follows from Equation (2).
- For output wires of multiplication gates the claim on the real value follows from Equation (7) and the claim on the external value follows immediately from the protocol.

5.1 Proof of Security

In this section we explain the security of our protocol in Section 3 and in Section 5.2 we give a brief overview on the security of our protocol in Section 4. The security considerations of our online protocol are very similar to those given in the proof of [21] for SPDZ online, and the security considerations of our protocol from Section 4 are very similar to the corresponding proof in Overdrive [31]. Thus, we mainly focus on the necessary changes.

We first prove the following security theorem for our function dependent offline,

[7]Note that due to the asymmetry in the multiplication, this is not possible if the value plays b in the other multiplication. In the SPDZ offline protocol of [21] there is no asymmetry in the multiplication and therefore this can be done more frequently.

Theorem 2. *In the $\mathcal{F}_{\text{Prep}}$-hybrid model, Protocol Π_{FDPrep} securely computes $\mathcal{F}_{\text{FDPrep}}$ in the presence of a static malicious adversary corrupting up to $n-1$ of the parties.*

*Proof Sketch.*We present the simulator $\mathcal{S}_{\text{FDPrep}}$ in Figure 5. For this simulator, we will assume the existence of a simulator $\mathcal{S}_{\text{Prep}}$ that

- Receives as input the number of input wires of each party $n_{I_1}, \ldots, n_{I_n}$, the number of multiplication gates n_M, and the number of additional random $[[\cdot]]$-shared elements n_R.
- Returns shares of
  - The global MAC α,
  - The input wire maskings $\mathcal{T}_{Input,1}, \ldots, \mathcal{T}_{Input,n}$,
  - The Beaver multiplication triples $\mathcal{T}_{Mult}$,
  - The additional random elements $\mathcal{T}_{Rand}$.

  And outputs the respective shares to the corresponding parties (with the parties also receiving the shared value on their input wires).

I.e., the simulator $\mathcal{S}_{\text{Prep}}$ basically simulates the SPDZ preprocessing functionality (see the description of the functionality, which is taken from [21], in Appendix D). Such a simulator is presented for example in [21, Appendix D].

The simulator $\mathcal{S}_{\text{FDPrep}}$ first runs $\mathcal{S}_{\text{Prep}}$ with the values required in the protocol (with $n_R = n_M$), then associates the input maskings to the input wires and the triples and random elements to the gates, and finally reveals the offsets as in the protocol. Security follows from observing that due to the MACCheck, the adversary can cheat on these offset values only with negligible probability.

$\square$

For the online phase protocol, we prove the following theorem

Theorem 3. *In the $\mathcal{F}_{\text{FDPrep}}$-hybrid model, the protocol Π_{Online} securely computes the function in the presence of a static malicious adversary corrupting up to $n-1$ of the parties.*

We first give some intuition on the security: the proof is very similar to the SPDZ protocol proof in [21]: We present a simulator $\mathcal{S}_{\text{Online}}$ (see Figure 7) on top of the ideal functionality $\mathcal{F}_{\text{Online}}$ (see Figure 6), such that the adversary cannot distinguish between interaction with the protocol Π_{Online} and the functionality $\mathcal{F}_{\text{FDPrep}}$ and interaction with $\mathcal{S}_{\text{Online}}$ and $\mathcal{F}_{\text{Online}}$. The ideal functionality $\mathcal{F}_{\text{Online}}$ is the same one as in [21], presented in Figure 6 for completeness.

The similarity to the proof of [21] is not surprising, because (ignoring the change of notation as explained in Section 2) the information seen by the adversary in our online protocol (in Section 3) is almost identical to the information seen in the online protocol of [21] – $\widehat{e}_x$ and $\widehat{e}_y$ in our protocol correspond to ρ and ϵ in SPDZ, and the offsets $\widetilde{\lambda_x}$ and $\widetilde{\lambda_y}$ can be seen as adding an additional random element λ and revealing the difference $a-\lambda$ (or $b-\lambda$). Clearly, from this the adversary can learn $a-a'$ (where a and a' are part of two different Beaver triples) if the same wire is input into multiple gates. However, observe that the difference $a-a'$ (for the same input wire in multiple gates) can also be computed in SPDZ by $\rho-\rho' = (a-v_x)-(a'-v_x) = a-a'$.

Conversely, observe that if $a-a'$ is revealed for every multiplication gate with the same input wire, then $a-\lambda$, $a'-\lambda$, etc., can be simulated by randomly choosing $a-\lambda$ for one of these gate input wires. Thus, simulation of our protocol can basically be seen as a simulation of SPDZ, with adding additional random elements and simulating them.[8]

In fact, the proof is almost identical to the online proof in [21]; the main difference is we need to show that the values revealed at the online phase, i.e., the external values, are uniformly random and independent of the offsets revealed at our function dependent offline phase.[9] We therefore begin by showing this observation:

[8]This explanation is given for intuitive purposes only – it is not fully accurate; a more precise statement is that for every fixing of the inputs, there exists a unique choice of a, a', λ, etc. that match the set of values seen by the adversary (i.e., the external values and the offsets).

[9]Recall that compared to the original SPDZ, in our case the adversary also receives the offsets $\widetilde{\lambda_x}, \widetilde{\lambda_y}$, revealed at our function dependent offline phase protocol, at each gate.

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

This functionality receives the circuit. The functionality does not receive any input from the parties.
Initialize: Call $\mathcal{F}_{\text{Prep}}$ with $n_{I_1}, \ldots, n_{I_n}$ the number of input wires (of parties $1,..,n$ respectively), n_M being the number of multiplication gates, n_S the number of square gates, , and $n_R = n_M$. The functionality receives the input wire maskings $\mathcal{T}_{Input,1}, \ldots, \mathcal{T}_{Input,n}$, the Beaver multiplication triples $\mathcal{T}_{Mult}$, and the additional random elements $\mathcal{T}_{Rand}$.
The functionality sets BreakDown flag according to $\mathcal{F}_{Prep}$. If $\mathcal{F}_{\text{Prep}}$ returned Abort, then the functionality aborts. Otherwise, the functionality forwards the relevant shares/values of the output of $\mathcal{F}_{\text{Prep}}$ to the corresponding parties.
Computation: Assuming BreakDown is false, the functionality associates with each input of party P_i an input masking $(r_i, [[r_i]]) \in \mathcal{T}_{Input,i}$. Then the functionality associates with each multiplication gate a Beaver triple $([[a]], [[b]], [[c]]) \in \mathcal{T}_{Mult}$ and a random element $[[r]] \in \mathcal{T}_{Rand}$, and sets $\lambda_c = c + r$.
Then, in topological order, on each addition gate sets with input wires x, y, and output wire z, sets $\lambda_z = \lambda_x + \lambda_y$.
At the end of this process, the functionality stores for each multiplication gate with input wires x, y the offsets $\widetilde{\lambda_x}$ and $\widetilde{\lambda_y}$.
Output: If BreakDown is false, the functionality outputs the offsets $\widetilde{\lambda_x}$ and $\widetilde{\lambda_y}$ for every multiplicationgate to the adversary, and waits for its reply

– If the adversary replies Abort, the functionality aborts,
– If the adversary replies Cheat, the functionality aborts, except with negligible probability[a] in which it sets BreakDown to true, forwards the shares of the honest parties to the adversary and sends the parties offsets $\widetilde{\lambda_x}'$, $\widetilde{\lambda_y}'$ chosen by the adversary.
– If the adversary replies Proceed, the functionality forwards the offsets $\widetilde{\lambda_x}, \widetilde{\lambda_y}$ to the parties.

If BreakDown is true, the functionality forwards the shares of the honest parties to the adversary and sends the parties offsets $\widetilde{\lambda_x}'$, $\widetilde{\lambda_y}'$ chosen by the adversary.

[a]This is the probability the adversary can cheat on the MACCheck protocol. From Claim 1, this is $\leq \frac{2}{|\mathbb{F}|}$.

Fig. 4: Offline phase functionality

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

Initialize: Run a local copy of the functionality $\mathcal{F}_{\text{Prep}}$ (i.e., run the simulator $\mathcal{S}_{\text{Prep}}$) with $n_{I_1}, \ldots, n_{I_n}$ the number of input wires in the circuit, n_M the number of multiplication gates, and $n_R = n_M$, receive all the shares of all the parties of the MAC α and of $\mathcal{T}_{Input,1}, \ldots, \mathcal{T}_{Input,n}$, $\mathcal{T}_{Mult}$, and $\mathcal{T}_{Rand}$.
If the environment inputs Abort then the functionality aborts. If the environment inputs Cheat the simulator simulates cheating in the protocol: if the cheat succeeds (happens with probability $\leq \frac{2}{|\mathbb{F}|}$) then all the simulated values for the honest parties are given to the adversary and BreakDown is set to true. Otherwise, the simulation aborts.
If the simulator did not abort, then it forwards the relevant shares/values to the corresponding parties.
Computation: Compute the offsets $\widetilde{\lambda_x}$ and $\widetilde{\lambda_y}$ for every input wire of a multiplication/squaring gate as in the protocol.
Finalization: All the offsets $\widetilde{\lambda_x}, \widetilde{\lambda_y}$ are given to the adversary and receives from the adversary $\widetilde{\lambda_x}'$, $\widetilde{\lambda_y}'$ (which might be different from $\widetilde{\lambda_x}, \widetilde{\lambda_y}$). Then the simulator simulates the MACCheck protocol on these offsets on behalf of all the honest parties.[a] If the check does not pass, then it aborts. Otherwise, it forwards all the offsets $\widetilde{\lambda_x}'$, $\widetilde{\lambda_y}'$ to the parties.

[a]Note that if BreakDown is already true, then the adversary knows α_i of all the parties and can therefore ensure these checks pass.

Fig. 5: The simulator for our offline phase

Observation 4 *Given any fixing of the offset values $\widetilde{\lambda_x}, \widetilde{\lambda_y}$ of all the gates, all the external values e_z of the output wires of multiplication gates are uniformly random in the view of the adversary.*

Proof Sketch. Let e_z be the external value of the output wire of a multiplication gate. We have that $e_z = v_z + \lambda_z = v_z + c + r$ where r is a fresh random. Thus, it is clear that e_z is not correlated with any external value and offset that appears before in a topological order on the circuit. By similar considerations, it is not correlated with the external value of the output wire of any proceeding multiplication/squaring gate, because these external values also have fresh randomness. Finally, this external value is not correlated to any input offset $\widetilde{\lambda_x}$ (or $\widetilde{\lambda_y}$) of a proceeding multiplication/squaring gate because the value of a (or b) of the triple in the respective gate, which is hidden, is independent of this external value. □

Notice that the above arguments would not have gone through, had we naïvely set $\lambda_\omega = c$, since c is correlated with a, b of the same gate. Thus, we could not argue that e_ω is independent of all previous external values and offsets. Indeed, we show in Appendix B that setting $\lambda_\omega = c$ results in an insecure protocol.

Remark 3. For our protocol in Section 4, Observation 4 is immediate since all the offset values are 0, and the permutation elements are uniformly random. Thus, for our protocol in Section 4, our main security task is to prove our modification to Overdrive is secure, i.e., the triples are generated correctly and no information on the permutation elements is leaked (which implies also no information is leaked on the multiplication triples beyond the correlations already known to be created by the protocol). This follows from the SPDZ sacrifice, and shown in Appendix 5.2.

We now turn to prove the security of our MPC protocol. We prove the following theorem:

Theorem 5. *In the $\mathcal{F}_{\text{FDPrep}}$-hybrid model, for any adversary $\mathcal{A}$ corrupting at most $n-1$ parties, the view of the adversary along with the outputs when running Π_{Online} in the real world is indistinguishable from the view of the adversary along with the outputs when interacting with $\mathcal{S}_{\text{Online}}$ on top of the ideal functionality $\mathcal{F}_{\text{Online}}$.*

Proof Sketch. If during the simulation the adversary cheats this is forwarded to $\mathcal{F}_{\text{Online}}$, and if the cheating is successful it outputs the inputs of the honest parties, which are then used to complete the simulation. Note that cheating at any step in the protocol implies that the adversary manages to later pass the MACCheck test with these values, and therefore by Claim 1 the probability of success in the simulation matches the probability in the real protocol. In the following we assume that the adversary did not attempt to cheat.

In the initialization, the simulator acts as $\mathcal{F}_{\text{FDPrep}}$. Then, for the input wires, the honest parties send the external values, which are uniformly random in the protocol due to the input maskings, and thus match the simulation. For addition gates, addition of scalar, and multiplication by a scalar, there is no interaction.

The interaction in multiplication and squaring gates is only a partial opening of the external value on the output wire the gate. By Observation 4, these are uniformly random, and thus match the simulation.

In the output stage, the environment sees the value v_y and the honest parties' shares which correspond to this value in both the protocol and the simulation. Again, if the value v_y in the real protocol is incorrect, it implies the adversary cheated and managed to pass the MACCheck, and therefore this happens with negligible probability. Thus, the value v_y in the real protocol is correct evaluation corresponding to the real inputs of the honest parties and matches the value v_y in the simulation. □

5.2 Security of Our Protocol in Section 4

In this section we give the ideas of the security proof for our protocol in Section 4. We stress that since we do not give the full details of the offline protocol (which generally follow Overdrive [31]) we cannot give a formal proof; this will be given in the full version. We therefore only attempt to point out the main changes needed for the corresponding proof in [31], in particular to the proof of [31, Theorem 2]. We use the definition of the $(\mathcal{F}_{[[]]}, \mathcal{F}_{\text{RAND}})$-hybrid model from [31].

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

Initialize: The functionality waits for an input command from the environment:

– If the input is Proceed, it sets BreakDown to false and continues.
– If the input is Abort, the functionality aborts.
– If the input is Cheat, with probability $\frac{2}{|\mathbb{F}|}$ it sets BreakDown to true. Otherwise, it proceeds as in Abort.

Input: On command $\mathsf{Input}(v_x)$ from P_i on its input wire, the functionality stores x in var, where var is a fresh identifier.
Add: On command $\mathsf{Add}(var_1, var_2)$ from the parties (with var_1, var_2 present in memory), the functionality retrieves var_1 and var_2 and stores $var_1 + var_2$ in var_3, where var_3 is a fresh identifier.
Multiply: On command $\mathsf{Multiply}(var_1, var_2)$ from the parties (with var_1, var_2 present in memory), the functionality retrieves var_1 and var_2 and stores $var_1 \cdot var_2$ in var_3, where var_3 is a fresh identifier.
Output: On command $\mathsf{Output}(\mathsf{var})$ (where the output var is present in memory), the functionality forwards the value of var, denoted v_y, to the environment.

– If BreakDown is false, the functionality waits for an input command from the environment. If the input is Proceed, then v_y is output to all the parties. Otherwise, ϕ is output to all the parties.
– If BreakDown is true, the functionality receives a false output v'_y from the environment and outputs it to all the parties.

Fig. 6: The ideal functionality for MPC

Theorem 6. *In the $(\mathcal{F}_{[[]]}, \mathcal{F}_{\text{RAND}})$-hybrid model, Protocol $\mathbf{\Pi_{FDTriples}}$ securely computes $\mathcal{F}_{\mathbf{FDTriples}}$ against a malicious adversary corrupting any number of parties.*

Proof Idea. The proof generally follows the proof of [31, Theorem 2]. The major difference is that now the shares of a and b are not randomly chosen in the protocol, but instead the shares of the permutation elements (of circuit input wires and multiplication gate output wires) are randomly chosen in the protocol. Nevertheless, since the shares of each a and b are a linear combination of these permutation elements, and since the shares of these permutation elements are given as input to $\mathcal{F}_{[[]]}$, the simulator can recover the handles of a and b needed for the simulation from the handles of the permutation elements which are received from the output of $\mathcal{F}_{[[]]}$.

A second subtle point is that in the proof of [31, Theorem 2], $\rho \leftarrow \mathbb{F}$ is sampled randomly by the simulator. Thus, we must ensure that although the definition of ρ has changed in $\mathbf{\Pi_{FDTriples}}$ (since $\rho = rb - \hat{b}$ and the definition of b has changed), it is still uniformly random assuming $\hat{b}$ and $\hat{c}$ are sacrificed. This follows since $\hat{b}$ is uniformly random and different for every revealed ρ.

Acknowledgements. We would like to thank Amos Beimel for helpful discussions. Special thanks to Ivan Damgård and Marcel Keller for helping us to understand SPDZ and Overdrive better.

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

Initialize: The simulation of the initialization procedure is performed running a local copy of $\mathcal{F}_{\text{FDPrep}}$. Observe that the data given to the adversary is known by the simulator, and that the simulator knows α. If the environment inputs Proceed, Cheat, or Abort to the copy of $\mathcal{F}_{\text{FDPrep}}$, the simulator does so to $\mathcal{F}_{\text{Online}}$ and forwards the output of $\mathcal{F}_{\text{Online}}$ to the environment: If the output is Success, the simulator sets BreakDown to true and uses the environment's inputs as preprocessed data. If $\mathcal{F}_{\text{Online}}$ outputs NoSuccess or the input was Abort, the simulator aborts.
Input: If BreakDown is false, honest input is performed according to the protocol, with a dummy input (e.g., all zeros).
If BreakDown is true, $\mathcal{F}_{\text{Online}}$ outputs the inputs of honest players, which can then be used in the simulation.
For inputs given by a corrupt party P_i, the simulator waits for P_i to broadcast the value e'_ω, computes $v'_\omega = e'_\omega - r_i$, where r_i is the input masking of that wire, and uses v'_ω as input to $\mathcal{F}_{\text{Online}}$. Observe that since the adversary is malicious, it may be that $v'_\omega \neq v_\omega$.
Add/Multiply: These procedures are performed according to the protocol. The simulator also calls the respective procedure in $\mathcal{F}_{\text{Online}}$.
Output: $\mathcal{F}_{\text{Online}}$ outputs v_y to the simulator.

- If BreakDown is false, the simulator now has to provide the honest parties' shares of such a value; it already computed an output value v'_y using the dummy inputs for the honest parties, so it can select a random honest party and modify its share adding $v_y - v'_y$ and modify the MAC adding $(v'_y - v_y)\alpha$ (recall that the simulator knows α). After that, the simulator opens v_y according to the protocol. If v_y passes the check, the simulator sends Proceed to $\mathcal{F}_{\text{Online}}$. Otherwise, it sends Abort.
- If BreakDown is true, the simulator inputs the result of the simulation to $\mathcal{F}_{\text{Online}}$.

Fig. 7: Simulator for the online phase

Functionality $\mathcal{F}_{\textbf{FDTriples}}$

Initialize: For every wire ω that is a circuit input wire or a multiplication gate output wire, randomly sample a permutation element $\lambda_\omega \leftarrow \mathbb{F}$.
In topological order on the circuit, for every addition gate with input wires x, y and output wire z, compute $\lambda_z = \lambda_x + \lambda_y$.

Triples: For every multiplication gate with input wires x, y and output wire z, set the multiplication triple of the gate to be $a = \lambda_x$, $b = \lambda_y$, and $c = a \cdot b$.

Output: Give the randomly chosen permutation elements and the computed c's as input to $\mathcal{F}_{[[]]}$ and output as it does.

Fig. 8: Functionality for generating triples correlated based on the circuit.

Bibliography

[1] B. Applebaum, Y. Ishai, E. Kushilevitz, and B. Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In CRYPTO, pages 166–184, 2013.

[2] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In ACM CCS, pages 805–817, 2016.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In IEEE SP, pages 843–862, 2017.

[4] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In EUROCRYPT, pages 483–501, 2012.

[5] M. Ball, T. Malkin, and M. Rosulek. Garbling gadgets for boolean and arithmetic circuits. In ACM CCS, pages 565–577, 2016.

[6] C. Baum, I. Damgård, and C. Orlandi. Publicly auditable secure multi-party computation. In SCN, pages 175–196. Springer, 2014.

[7] C. Baum, I. Damgård, T. Toft, and R. Zakarias. Better preprocessing for secure multiparty computation. In ACNS, pages 327–345. Springer, 2016.

[8] D. Beaver. Efficient multiparty protocols using circuit randomization. In CRYPTO, pages 420–432, 1991.

[9] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In STOC, pages 503–513,1990.

[10] A. Ben-Efraim. On multiparty garbling of arithmetic circuits. In ASIACRYPT, pages 3-33, 2018.

[11] A. Ben-Efraim, Y. Lindell, and E. Omri. Optimizing semi-honest secure multiparty computation for the internet. In ACM CCS, pages 578–590, 2016.

[12] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In STOC, pages 1–10, 1988.

[13] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In EUROCRYPT, pages 169–188, 2011.

[14] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In STOC, pages 11–19, 1988.

[15] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In CT-RSA, pages 416–432, 2012.

[16] R. Cramer, I. Damgrd, D. Escudero, P. Scholl, and C. Xing. SPDZ2k: efficient MPC mod 2^k for dishonest majority. In CRYPTO, 2018.

[17] I. Damgård and Y. Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In CRYPTO, pages 378–394, 2005.

[18] I. Damgård and Y. Ishai. Scalable secure multiparty computation. In CRYPTO, pages 501–520, 2006.

[19] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In CRYPTO, pages 572–590, 2007.

[20] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In EUROCRYPT, pages 445–465, 2010.

[21] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In CRYPTO, pages 643–662, 2012.

[22] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In ESORICS, pages 1–18, 2013.

[23] I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In CRYPTO, pages 167–187, 2017.

[24] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In STOC, pages 218–229, 1987.

[25] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In ASIACRYPT, pages 59–85, 2018.

[26] C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In, ASIACRYPT, pages 598–628, 2017.

[27] M. Hirt and J. B. Nielsen. Robust multiparty computation with linear communication complexity. In CRYPTO, pages 463–482, 2006.

[28] M. Hirt, U. Maurer, and B. Przydatek. Efficient secure multi-party computation. In ASIACRYPT, pages 143–161, 2000.

[29] M. Keller, E. Orsini, and P. Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In ACM CCS, pages 830–842, 2016.

[30] M. Keller, E. Orsini, D. Rotaru, P. Scholl, E. Soria-Vazquez, and S. Vivek. Faster secure multi-party computation of aes and des using lookup tables. In ACNS, pages 229–249, 2017.

[31] M. Keller, V. Pastro, and D. Rotaru. Overdrive: making spdz great again. In EUROCRYPT, pages 158–189, 2018.

[32] J. Kilian. Basing cryptography on oblivious transfer. In STOC, pages 20–31, 1988.

[33] E. Larraia, E. Orsini, and N. P. Smart. Dishonest majority multi-party computation for binary circuits. In CRYPTO, pages 495–512, 2014.

[34] Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In CRYPTO, pages 319–338, 2015.

[35] T. Malkin, V. Pastero, and a. shelat. An algebraic approach to garbling. Unpublished manuscript.

[36] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In STOC, pages 73–85, 1989.

[37] G. Spini and S. Fehr. Cheater detection in SPDZ multiparty computation. In ICITS, pages 151–176, 2016.

[38] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In ACM CCS, pages 39–56,2017.

[39] A. C. Yao. Protocols for secure computations. In FOCS, pages 160–164, 1982.

A Addition and Multiplication by a Public Scalar

In [22, 21] there is no explicit discussion on addition and multiplication by a constant at the online phase. Apparently, this is due to the simplicity of these operations there, i.e., this is easily achieved there by Property 1. However, in our protocol this requires slightly more care, because these operations also affect the external value, and possibly also the permutation element. Therefore, for completeness, we next elaborate on these operations for our protocol in Section 3. Our protocol in Section 4 also requires some small modifications to handle these operations – due to the similarity, these are omitted.

Formally, we treat these operations as unary gates, with input wire ω and output wire ω', as follows:

1. To add a public constant c, at the online phase the parties locally compute
   - $[[v_{\omega'}]] \leftarrow [[v_\omega]] + c$ using Property 1, and
   - $e_{\omega'} \leftarrow e_\omega + c$.

   Note that there is no need to modify the permutation element, i.e., $\lambda_{\omega'} = \lambda_\omega$, since $e_{\omega'} = e_\omega + c = v_\omega + \lambda_\omega + c = v_{\omega'} + \lambda_{\omega'}$ as required. This implies that addition by a public constant requires no modification at the offline phase.
2. To multiply by a public constant c, at the online phase the parties locally compute
   - $[[v_{\omega'}]] \leftarrow c \cdot [[v_\omega]]$ using Property 1, and
   - $e_{\omega'} \leftarrow c \cdot e_\omega$.

   Note that in this case, the parties also need to modify $[[\lambda_{\omega'}]] \leftarrow c \cdot [[\lambda_\omega]]$ at the offline phase, because $e_{\omega'} = c \cdot e_\omega = c \cdot (v_\omega + \lambda_\omega) = c \cdot v_\omega + c \cdot \lambda_\omega = v_{\omega'} + c \cdot \lambda_\omega$ should be equal to $v_{\omega'} + \lambda_{\omega'}$.

Remark 4. We note that at these specific gates, i.e., addition of a public constant and multiplication by a public constant, our online protocol requires slightly more operations than SPDZ, because in our protocol 2 values are modified whereas in SPDZ only a single value is modified. While addition is usually considered fast, multiplication is considerably slower. Thus, if the circuit contains many multiplications by a public constant in comparison with the number of regular multiplication gates, our online protocol might not be *computationally* superior to SPDZ. However, these operations require no communication, so our online protocol is always significantly superior in terms of *communication*.

B The Problem with Setting $\lambda_\omega = c$

We now show, using a simple example, that naïvely setting $\lambda_\omega = c$ in output wires of multiplication/squaring gates would be insecure. The example we chose is for simplicity and is not general, but a more general example can also be constructed. The example includes a squaring gate, the details of which were omitted from the main text – in the online phase, a squaring gate is computed using a square pair via Equation 8: The parties locally compute the output wire shares $[[v_z]] \leftarrow (\widehat{e}_x)^2 - 2 \cdot \widehat{e}_x[[a]] + [[c]]$ (where z is the output wire and x the input wire), and then set $[[e_z]] = [[v_z]] + [[\lambda_z]]$ and partially reveal $[[e_z]]$ as usual.

Example 1. We assume the circuit contains a square gate with input wire x and output wire z, which is not an output wire of the circuit. We further assume that the adversary has partial information on the data of the wire x and knows the real value is a bit, i.e., either 0 or 1.[10] Notice that from the squaring gate part of the online protocol and from Equation 5, the adversary learns $\widehat{e}_x = e_x + \widetilde{\lambda_x} = v_x + a$ and $e_z = v_z + \lambda_z$. Since now $\lambda_z = c = a^2$, it implies the adversary learns $v_x + a$ and $v_z + \lambda_z = v_x^2 + a^2 = v_x + a^2$ (recall that we assume $v_x \in \{0,1\}$, so $v_x^2 = v_x$), and compute $(v_x + a^2) - (v_x + a) = a^2 - a$. Therefore, unless $a \in \{0,1\}$, the adversary can deduce a from $a^2 - a$ and $v_x + a$ and thus learn v_x.

In the case that $\lambda_z = c + r$, the above method does not work, since the extra randomness introduced ensures the values $\widehat{e}_x$ and e_z are independently random, as required in the proof.

[10]While this is not the standard scenario in secure computation, also in this case a secure protocol should not allow the adversary to learn more than it already knows.

C MACCheck Protocol

For the protocol MACCheck, we assume the parties have access to an ideal commitment functionality $\mathcal{F}_{\text{Commit}}$, which allows a party to commit to a field element without revealing any information on it, and later open (only) this commited value to all the parties. We also assume the parties have access to an ideal random functionality $\mathcal{F}_{\text{RAND}}$, which allows the parties to jointly sample uniformly random elements $r \in \mathbb{F}$. The MACCheck protocol is formally presented in Figure 9.

Protocol MACCheck

CheckOutput($a_1, \ldots, a_t$)**:**

1. The parties sample random elements $r_1, \ldots, r_t \in \mathbb{F}^t$ using $\mathcal{F}_{\text{RAND}}$.
2. Each party i computes the linear combination $\sigma_i = \Sigma_{j=1}^{t} r_j(\gamma(a_j)_i - \alpha_i \cdot a_j)$.
3. The parties commit to their values of σ_i using $\mathcal{F}_{\text{Commit}}$.
4. The committed values are opened to all the parties using $\mathcal{F}_{\text{Commit}}$.
5. The protocol outputs Proceed if $\Sigma_{i=1}^{n} \sigma_i = 0$, otherwise outputs Abort.

Fig. 9: Protocol to check validity of MACs

As stated in Claim 1, if the adversary tries to cheat, the protocol outputs Proceed with probability $\leq \frac{2}{|\mathbb{F}|}$, which is negligible in the security parameter. See [21] for the proof.

D SPDZ Preprocessing Functionality

For completeness, we provide here the SPDZ preprocessing functionality given in [21] (with slight modifications).

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

Initialize: On input Start from honest parties and adversary, the functionality sets the internal flag BreakDownto false and then performs the following:

1. For each corrupted player $i \in \mathcal{A}$, the functionality accepts shares α_i from the adversary, and it samples at random α_i for each honest party $i \notin \mathcal{A}$. The functionality sets $\alpha = \Sigma_{i=1}^{n} \alpha_i$.
2. The functionality waits for command Abort, Proceed or Cheat from the adversary.
3. If received Proceed, the functionality outputs α_i to party i.
4. Otherwise, and if the functionality did not abort in Cheat, it outputs adversary's contribution α_i' to party i.

Computation: On input DataGen from all honest players and adversary, if the functionality received Proceed in initialize (or if BreakDown is true) it executes the data generation procedures specified in Figure 11:

– **Input Production** is executed n_{I_i} times for each party i,
– **Multiplication Triple** is executed n_M times,
– **Square Pair** is executed n_S times,
– **Random Element** is executed n_R times.

Cheat: The functionality chooses to do either one of the following:

– It sends, with probability $\leq \frac{2}{|\mathbb{F}|}$, Success to the adversary and sets the internal flag BreakDown to true.
– Otherwise it sends NoSuccess to the adversary and players, and goes to Abort.

Abort: The functionality outputs ϕ to all parties.

Fig. 10: The SPDZ preprocessing functionality

Functionality $\mathcal{F}_{\text{Prep}}$, Continuation

Macro $\textsf{\textbf{Angle}}(\mathbf{a_1}, \ldots, \mathbf{a_n}; \boldsymbol{\Delta}_\gamma)$**:** This macro is run by the following functions to generate a $[[\cdot]]$-shared elements. Denote $a = \Sigma_{i=1}^{n}$; the functionality does the following:

1. Receives $\{\gamma(a)_i\}_{i \in \mathcal{A}}$ from the adversary.
2. Sets $\gamma(a) = a \cdot \alpha + \Delta_\gamma$ and samples $\{\gamma(a)_i\}_{i \notin \mathcal{A}}$ at random, subject to $\gamma(a) = \Sigma_{i=1}^{n}\gamma(a)_i$.
3. Returns $(\gamma(a)_1, \ldots, \gamma(a)_n)$.

In all the following procedures, the output is assuming $\mathsf{BreakDown}$ is false. If $\mathsf{BreakDown}$ is true, then the outputs sent to the parties are set by the adversary.
Input Production: to generate an input masking, the functionality performs the following:

1. If $\mathsf{BreakDown}$ is true and the input is of an honest party, chooses a random value $r \in \mathbb{F}$. Otherwise, receive the value r from the adversary.
2. Receives from the adversary corrupted shares $(r)_i$ for $i \in \mathcal{A}$, and an offset Δ_γ for the MAC value.
3. Samples $(r)_i$ for $i \notin \mathcal{A}$ randomly, under the constraint that $r = \Sigma_{i=1}^{n}(r)_i$.
4. Runs macro $\mathsf{Angle}((r)_1, \ldots, (r)_n; \Delta_\gamma)$.
5. Outputs $((r)_i, \gamma(r)_i)$ to party i for each $i \in [n]$ and r to the party of the input wire.

Multiplication Triple: to generate a Beaver multiplication triple, the functionality performs the following:

1. Receives from the adversary corrupted shares $(a)_i$, $(b)_i$, and $(c)_i$ for $i \in \mathcal{A}$ and offsets $\Delta_{\gamma,a}, \Delta_{\gamma,b}$, and $\Delta_{\gamma,c}$.
2. Uniformly samples $(a)_i$, $(b)_i$, and $(c)_i$ for $i \notin \mathcal{A}$, under the constraint that $\Sigma_{i=1}^{n}c_i = (\Sigma_{i=1}^{n}a_i)\cdot(\Sigma_{i=1}^{n}b_i)$.
3. Runs macro $\mathsf{Angle}((a)_1, \ldots, (a)_n; \Delta_{\gamma,a})$, macro $\mathsf{Angle}((b)_1, \ldots, (b)_n; \Delta_{\gamma,b})$, and macro $\mathsf{Angle}((c)_1, \ldots, (c)_n ;\Delta_{\gamma,c})$.
4. Outputs $((a)_i, \gamma(a)_i)$, $((b)_i, \gamma(b)_i)$, and $((c)_i, \gamma(c)_i)$ to party i for each $i \in [n]$.

Square Pair: to generate a square pair, the functionality performs the following:

1. Receives from the adversary corrupted shares $(a)_i$ and $(c)_i$ for $i \in \mathcal{A}$ and offsets $\Delta_{\gamma,a}, \Delta_{\gamma,c}$.
2. Uniformly samples $(a)_i$ and $(c)_i$ for $i \notin \mathcal{A}$, under the constraint that $\Sigma_{i=1}^{n}c_i = (\Sigma_{i=1}^{n}a_i)^2$.
3. Runs macro $\mathsf{Angle}((a)_1, \ldots, (a)_n; \Delta_{\gamma,a})$ and macro $\mathsf{Angle}((c)_1, \ldots, (c)_n; \Delta_{\gamma,c})$.
4. Outputs $((a)_i, \gamma(a)_i)$ and $((c)_i, \gamma(c)_i)$ to party i for each $i \in [n]$.

Random Element: to generate a random $[[\cdot]]$-shared element, the functionality performs the following:

1. Receives from the adversary corrupted shares $(r)_i$ for $i \in \mathcal{A}$ and an offset Δ_γ,
2. Uniformly samples $(r)_i$ for $i \notin \mathcal{A}$,
3. Runs macro $\mathsf{Angle}((r)_1, \ldots, (r)_n; \Delta_\gamma)$.
4. Outputs $((r)_i, \gamma(r)_i)$ to party i for each $i \in [n]$.

Fig. 11: The SPDZ preprocessing functionality auxilary functions