Use your Brain! Arithmetic 3PC For Any Modulus with Active Security*

Hendrik Eerikson[1], Marcel Keller[4], Claudio Orlandi[2], Pille Pullonen[1], Joonas Puura[3], Mark Simkin[2]

[1] {hendrik.eerikson, pille.pullonen}@cyber.ee, Cybernetica AS
[2] {orlandi,simkin}@cs.au.dk, Department of Computer Science, DIGIT, Aarhus University
[3] University of Tartu (work done while at Cybernetica AS)
[4] mks.keller@gmail.com, CSIRO's Data61

Abstract. Secure multiparty computation (MPC) allows a set of mutually distrustful parties to compute a public function on their private inputs without revealing anything beyond the output of the computation. In recent years, a large effort has undergone into designing and implementing MPC protocols that can be used in practice. This paper focuses on the specific case of actively secure three-party computation with an honest majority, which is among the most popular models for real-world applications of MPC. In particular, we are interested in solutions which allow to evaluate arithmetic circuits over real-world CPU word sizes, like 32- and 64-bit words, that are secure against active adversaries that may arbitrarily deviate from the protocol description.
This paper presents better protocols for this important setting. Our starting point is the novel compiler of Damgård et al. from CRYPTO 2018, and we improve it in several ways: First, we present an improved version of their compiler which reduces the online communication complexity by a factor of 2. Next, we replace their preprocessing protocol (which performs arithmetic modulo a large prime) with a more efficient preprocessing which only performs arithmetic modulo powers of two (and is therefore more efficient). And finally, we present a novel protocol which replaces the preprocessing phase with a "postprocessing" check. The protocols we construct offer different efficiency tradeoffs and can therefore outperform each other in different deployment settings.
We also provide a fully-fledged implementation of our protocols, and extensive benchmarks of our framework in a LAN and in different WAN settings. Concretely, we achieve a throughput of 3 million 64-bit multiplications per second with each of the three parties located on a different continent and 10 million in the same location, thus achieving the most effcient implementation of a three-party computation protocol for arithmetic circuits modulo 2^{64} with active security.

"Brain, what do you want to do tonight?",
"The same thing we do every night, Pinky - try to take over the world!"

Pinky and the Brain

***Summary of Changes**: This version of the preprint contains two new significantly more efficient protocols (Sections 6.2 and 7) and new benchmarks and implementations (Section 8). In addition, Marcel Keller was added to the list of authors.*

1 Introduction

Secure Multiparty Computation (MPC) is an umbrella term for a broad range of cryptographic techniques and protocols that enable a set of parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$ to compute some function f of their private inputs

* This work was supported by the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC), the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation program under grant agreement No 669255 (MPCPRO), No 731583 (SODA), No 803096 (SPEC), No 778615 (BiggerDecisions), and Estonian Research Council under grant IUT27-1.

$x_1, \ldots, x_n$ without revealing anything beyond the output $f(x_1, \ldots, x_n)$ of the computation. Most importantly, an actively misbehaving participant of the computation should not be able to bias the outcome of the computation (except by choosing their input) or learn anything about the inputs of the honest parties (except for what is leaked by the output itself). MPC has started out as a purely theoretical research field in the 90ies, but has recently developed into a science on the brink of practical deployment. This is witnessed by the constantly increasing number of real-world use cases, MPC framework implementations, and startups (see [ABL+18] for a survey).

The landscape of MPC protocols is broad and diverse, and protocols differ greatly depending on many parameters such as the number of involved parties, the corruption threshold, the adversarial model, and the network setting.

In this paper we focus on one of the most popular models for MPC, namely *three-party computation with an honest majority*. This model has been used in different real-world applications of MPC [BCD+09, BTW12, BJSV15, BKK+16, AKTZ17], often in the so-called *client-server* scenario where a possibly large number of clients secret share their inputs to three computation servers who can then perform the desired computation securely on their behalf [JNO14] and return the result to the clients. A major advantage of the honest majority setting is that one can obtain protocols which do not rely on computationally expensive cryptographic operations such as exponentiations, oblivious transfer, etc, but typically only use light-weight arithmetic operations.

Existing implementations of three-party computation protocols for the honest-majority case fall into two broad categories: VIFF [DGKN09] and its successors [Sch18] only support arithmetic computations over prime order fields. Sharemind's protocol suite [BLW08, BNTW12] can be used to evaluate arithmetic circuits with arbitrary word sizes, but is only secure against passive adversaries that follow the protocol faithfully. In practice this means that one has to either settle for rather weak security guarantees or one has to develop applications specifically tailored to rather unnatural word sizes instead of using the common 32- and 64-bit word sizes that dominate real-world system architectures. In particular, this means that a developer has to match the needs of the MPC framework it wants to use rather than the MPC framework matching the needs of the developer.

The main barrier to constructing actively secure protocols for evaluating arithmetic circuits with arbitrary word sizes lies in the fact that known approaches to achieving active security, like *information checking* techniques [RB89], require prime order fields. Up until recently it has been an open question to design protocols for arithmetic circuits with active security for arbitrary word sizes. In a recent work Damgård et al. [DOS18] addressed this question by presenting a protocol compiler that transforms passively secure protocols into actively secure ones that can tolerate up to $\mathcal{O}(\sqrt{n})$ corruptions and only have a *constant* overhead in storage and computational work.

Our Contributions. In this paper we consider the class of protocols produced by compiler of Damgård et al. [DOS18], and we improve such protocols in several ways. The main idea behind Damgård et al.'s compiler is to let the *real parties* "emulate" *virtual parties* that execute the desired computation on behalf of the real parties. The crucial point behind their compiler is that the virtual parties can execute[5] a passively secure protocol in a way that prevents any real party from actively misbehaving. Every time that a virtual party $\mathbb{P}_i$ is supposed to send a message to another virtual party $\mathbb{P}_j$ in the passively secure protocol, every real party that is emulating $\mathbb{P}_i$ computes the same message redundantly and sends it to every real party which is emulating $\mathbb{P}_j$. Each real party emulating $\mathbb{P}_j$ therefore receives a set of messages and aborts in case the received messages are not all equal. Intuitively this approach ensures active security as long as there is at least one honest party in the emulating set of every virtual party, since any malicious party either follows the protocol (in which case we effectively only have passive corruptions) or sends a message that disagrees with the message that is sent by at least one honest party (in which case the honest receiving party and consequently all other parties abort the protocol). This approach heavily relies on the fact that *all* protocol

[5] Note that virtual parties do not physically exist. Whenever we say that "virtual parties execute a protocol", we really mean that the real parties simulate the virtual parties that execute the protocol.

messages are sent redundantly, thus incurring a multiplicative blow-up in the bandwidth overhead of the protocol.

Our first contribution is of theoretical nature: we present an improved compiler that significantly reduces the number of redundant messages that need to be sent during a protocol execution. The idea behind our approach is to elect *one* real party in each virtual party to be the "brain", which sends all messages on behalf of its virtual party to *all* real parties in the receiving virtual party. The other real parties, the "pinkies", still receive messages from the brains and thus can locally follow the protocol execution. At the end of the protocol, right before the output of the computation is released, we then let all parties perform a single check that guarantees that all messages which were sent by the brains during the protocol are consistent with the messages all the pinkies would have sent. This check can be performed very efficiently by only checking consistency of the *hashes* of the protocol transcripts. It is clear that if any of the brains cheated during the protocol execution, then it must have sent a message that is inconsistent with the view of at least one pinky, thus the protocol would abort during the checking phase. On the downside our new compiler now imposes a stronger security requirement on the protocol it starts with. Note that honest brains continue the protocol execution up to the checking phase even if a malicious brain misbehaves, which means that we need a protocol that does not leak any private information even if cheating during the computation phase occurs. Thankfully, most passively secure secret sharing based protocols provide exactly the security guarantees that we need. More concretely, these protocols follow a compute-then-open structure, where the output of the computation is only revealed in the last round and any cheating during the preceding computation rounds can only affect the correctness of the output, but not the privacy of the inputs. Thus, by performing the consistency check described above at the end of the computation phase and *before* the output phase, we can ensure that no information is leaked. The security property sketched above has previously appeared in the literature under the name of weak privacy [GIP+14].

We formally present our new compiler and prove its security in Section 4. For the specific three-party case, our compiler produces a protocol, which is roughly twice as efficient as the protocol produced by the compiler of Damgård et al., since in the three party case each virtual party is emulated by one pinky and one brain.

Our second contribution is an improved preprocessing protocol for generated secret-shared multiplication triples. Damgård et al. generate both triples modulo a prime and triples modulo a power of 2, followed by a check-and-sacrifice step. We replace this by a preprocessing phase which does not perform any arithmetic in the larger prime field and solely uses computation modulo a slightly larger power of 2, thus improving on efficiency. While the sacrifice step is not performed in a field anymore, security follows using similar arguments as in the recent work on SPDZ over rings [CDE+18].

Finally we show that it is possible to completely avoid the preprocessing phase if one wishes to do so. Recall that our underlying protocols are assumed to preserve privacy until the outputs are opened. We exploit this security property by running the multiplication protocols optimistically and then, prior to opening the outputs, perform a single combined check. The two protocols, with preprocessing and with the postprocessing check, therefore offer different efficiency tradeoffs. The protocol with preprocessing has a leaner online phase, whereas the protocol with postprocessing has a better overall performances. Descriptions of our protocols are given in Sections 5, 6, 7.

In Section 8, we provide extensive performance benchmarks of our framework, both in the LAN as well as different WAN settings. Our protocols have been integrated in two of the leading MPC frameworks, namely the Sharemind MPC protocol suite and MP-SPDZ. As described in Section 8, we achieve the most efficient implementation of a three-party computation protocol for arithmetic circuits modulo 2^{64} with active security.

Other Related Work. The SPDZ family of protocols [BDOZ11, DPSZ12, DKL+13] efficiently implements MPC with active security in the *dishonest majority* setting. These protocols are split up into a slower, computationally secure *offline phase* in which correlated randomness in the form of so-called Beaver's triples is generated and a faster, information-theoretically secure *online phase* in which these triples are consumed to compute the desired functionality. Active security in the online phase is achieved using information theoretic message authentication codes (MACs), which until recently limited the SPDZ approach to computation over fields. In a recent work of Cramer et al. [CDE+18], this limitation has been lifted, allowing to perform

computation modulo 2^k (by defining the MACs modulo to be $2^{k+\lambda}$ where λ is the security parameter, thus introducing an overhead proportional to the security parameter). An implementation (and optimizations) of [CDE+18] was presented in [DEF+19]. In addition [CRFG19] follows up [CDE+18] with a two-party protocol that uses homomorphic encryption and efficient zero-knowledge proofs in the precomputation phase.

Other recent works have considered active security in the three-party setting: [FLNW17] uses correlated random number generation to achieve efficient preprocessing and replication to achieve active security. The protocol was originally presented only for Boolean circuits, but it was then later noticed that the approach generalizes to general rings [MR18]. They mention actively secure protocols in this setting, but do not give detailed protocol descriptions and only implement semi-honest versions of their protocols. For finite fields, [CGH+18] achieves active security by running two copies of the computation, respectively with real and random inputs, and uses the latter to verify correctness (their approach can also be used for more than three parties).

After the first version of our paper appeared online, a very different protocol for the same three party honest majority setting was presented in [CCPS19]. They combine two linear secret sharing schemes, one between two and other between three parties where the former is used to share a component of the latter sharing. This allows them to create a circuit dependent precomputation phase where all the two party sharings of random values are precomputed based on the circuit structure. The online phase focuses on computing modifiers to turn the random precomputed sharings into the desired values. Moreover, novel techniques for honest-majority MPC over rings have very recently been deployed in [ACD+19]. It is however still unclear whether this can lead to protocols which are efficient in practice.

2 Preliminaries

We write $v \leftarrow \mathcal{X}$ to denote the sampling of a uniformly random value v from set $\mathcal{X}$. Throughout the paper λ denotes the security parameter. Given n parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$, we write P_{i+1} to denote the party after P_i and we implicitly assume a wrap around of the party's index. That is $\mathrm{P}_{n+1} = \mathrm{P}_1$ and $\mathrm{P}_0 = \mathrm{P}_n$.

2.1 Security Definitions

3 Security Definitions

We define security using the UC framework of Canetti [Can01]. Protocols proven secure in this framework retain security even when composed arbitrarily and executed concurrently. Concretely, we will use a flavour of the classical UC framework, which was proposed in [CDN15]. We will provide a short summary of the security framework here and refer the reader to [CDN15] for more details.

Security is defined by comparing a real and an ideal interaction. In the ideal interaction, we have a trusted party, called the ideal functionality $\mathcal{F}$, that receives inputs from all parties, computes some desired function on those inputs, and returns the result to the parties. In the real interaction, the parties do not have access to $\mathcal{F}$, but rather interact with each other according to some protocol description Π. The protocol Π may itself make use of some other auxiliary ideal functionality $\mathcal{G}$. In both interactions, the environment $\mathcal{Z}$ chooses the inputs of all parties and acts as an adversary that may corrupt some subset of the parties passively or actively. We say that Π securely realizes $\mathcal{F}$ if an adversary in the real world can not do "more harm" than an adversary in the ideal world. Concretely, we require the existence of a simulator $\mathcal{S}$, called ideal world adversary, that simulates $\mathcal{Z}$'s view of a real interaction. $\mathcal{S}$ simulates the views of the corrupted players, the interaction with auxiliary functionality $\mathcal{G}$, and it may itself interact with $\mathcal{F}$. At the end of a protocol execution $\mathcal{Z}$ outputs a single bit. Let $\mathsf{IDEAL}_\lambda[\mathcal{Z}, S, \mathcal{F}]$ and $\mathsf{REAL}_\lambda[\mathcal{Z}, \Pi, \mathcal{G}]$ be the random variables that represent $\mathcal{Z}$'s output bit in the ideal and real execution, respectively. We say that Π securely realizes functionality $\mathcal{F}$, if $\mathcal{Z}$ cannot distinguish whether it was part of a real interaction or whether it was communicating with the simulator $\mathcal{S}$.

Definition 1. *Π securely implements functionality $\mathcal{F}$ with respect to a class of environments Env in the $\mathcal{G}$-hybrid model, if there exists a simulator S such that for all $\mathcal{Z} \in Env$ we have*

$$|\Pr[\mathsf{REAL}_\lambda[\mathcal{Z}, \Pi, \mathcal{G}] = 1] - \Pr[\mathsf{IDEAL}_\lambda[\mathcal{Z}, S, \mathcal{F}] = 1]| \leq \mathsf{negl}(\lambda)$$

Using this definition we can now capture different security notions by different classes of environments.

Passive security. The environment $\mathcal{Z}$ can corrupt up to t parties. $\mathcal{Z}$ gets full read-only access to the corrupted parties internal tapes. All parties follow the protocol honestly. The simulator $\mathcal{S}$ is allowed to ask the ideal functionality $\mathcal{F}$ for the inputs of the corrupted parties.

Active security. The environment $\mathcal{Z}$ is allowed to corrupt up to t parties. $\mathcal{Z}$ gets full control of the corrupted parties. Once the ideal functionality $\mathcal{F}$ received inputs from all parties, it computes the output and sends it to $\mathcal{Z}$. The environment sends back a bit to $\mathcal{F}$ indicating whether the parties should obtain the output or $\perp$. A slightly weaker notion known as active security with individual abort allows the adversary to specify which honest parties abort and which do not.

Active Security with "Weak Privacy". We use the definition of active security with weak privacy [GIP+14, Definition 5.11] (a slight variant of the same property was defined under the name "active privacy" in [PL15]), which captures the security properties offered by many existing protocols [BGW88, Bea92] that follow the compute-then-open paradigm. These protocols are split into a computation and opening phase. The computation phase consists of multiple rounds of interaction, whereas the opening phase requires a single round of communication. Intuitively, weak privacy says that an active adversary cannot learn anything until the opening phase, and this is captured saying that there exists a simulator that can simulate the truncated view of the protocol up to the opening phase without having access to the inputs or outputs of any honest parties. Finally, these protocols are "linear", meaning that the output of the parties in the protocol is a linear function of the messages sent in the opening phase.

Throughout the paper, we assume a synchronous communication network, a rushing adversary, and secure point-to-point channels.

3.1 Auxiliary Ideal Functionalities

We will make use of the following basic auxiliary ideal functionalities in this paper: The broadcast with *individual* abort functionality $\mathcal{F}_{\mathsf{bcast}}$ (Figure 1) allows a sender S to send a value v to a set of parties $\mathbb{P}$. The functionality guarantees that either a party aborts or it agrees on a consistent value with the other parties. Such a functionality is weaker than detectable broadcast [FGMv02], which requires that either all players agree on the same value or that all players unanimously abort. The functionality can easily be instantiated by letting the sender S send v to all parties in $\mathbb{P}$. Every party in $\mathbb{P}$ echoes the received value to all other parties in $\mathbb{P}$. Parties that receive consistent values output that value, parties that receive inconsistent values abort.

The message checking functionality $\mathcal{F}_{\mathsf{check}}$ (Figure 2) allows a receiver, who holds a vector of messages, to check whether all other parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$ hold the same vector of messages. The functionality can be instantiated by letting each party P_i sends its input to R. However, in this case the communication overhead would be $\Theta(n\ell)$ messages, where ℓ is the number of messages in a vector. Assuming the existence of collision-resistant hash functions, one can obtain a more communication efficient solution by simply letting all parties hash their message vectors into small digests before sending them to R. The communication overhead of such a solution would be $\Theta(n\lambda)$ bits if we assume that the output length of the hash function is $\Theta(\lambda)$.

3.2 Additive Secret Sharing

We recall what additive secret sharing is and how to perform some basic operations on it. We will use this type of secret sharing in our three-party protocol in Section 5 and the modulus 2^m defines the word size over

Functionality $\mathcal{F}_{\mathsf{bcast}}$

The functionality runs with sender S, who has input v, parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$, and adversary $\mathcal{A}$.

---

1. S sends $(v, \mathbb{P})$ to $\mathcal{F}_{\mathsf{bcast}}$, where $v \in \{0,1\}^*$ and $\mathbb{P} \subset \{\mathrm{P}_1 \ldots \mathrm{P}_n\}$.
2. If either S or a party from $\mathbb{P}$ is corrupt, then $\mathcal{A}$ receives v and can decide which parties from $\mathbb{P}$ abort and which receive the output by sending a $|\mathbb{P}|$ long bit-vector b to the ideal functionality. For $\mathrm{P}_i \in \mathbb{P}$:
   (a) If $b_i = 1$, then $\mathcal{F}_{\mathsf{bcast}}$ sends v to P_i.
   (b) If $b_i = 0$, then $\mathcal{F}_{\mathsf{bcast}}$ sends $\perp$ to P_i.

Fig. 1: Broadcast functionality

Functionality $\mathcal{F}_{\mathsf{check}}$

The functionality runs with receiver R, parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$, and adversary $\mathcal{A}$. Party $\mathrm{P}_i \in \{\mathrm{P}_1, \ldots, \mathrm{P}_n\}$ has input $\left(m_{(1,i)}, \ldots, m_{(\ell,i)}\right)$ and receiver R has input $(m_1, \ldots, m_\ell)$.

---

1. All parties send their inputs to the ideal functionality.
2. $\mathcal{A}$ can decide whether to continue or to abort.
   (a) If $\mathcal{A}$ continues, then $\mathcal{F}_{\mathsf{check}}$ checks whether all message vectors are the identical. It outputs same if this is the case, and $\mathsf{different}$ otherwise, to the receiver R (in the latter case, the functionality outputs the inputs of all honest parties to $\mathcal{A}$).
   (b) If $\mathcal{A}$ aborts, then $\mathcal{F}_{\mathsf{check}}$ sends $\perp$ to all parties.

Fig. 2: Message checking functionality

which computations will be performed. For example, for arithmetic computations over 64-bit integers, one can set $m = 64$. For the sake of concreteness, we restrict our attention to the three-party case.

Sharing a value: If party P_i wants to share a value $a \in \mathbb{Z}_{2^m}$, it picks uniformly random $a_1, a_2 \leftarrow \mathbb{Z}_{2^m}$, sets $a_3 = a - a_1 - a_2 \mod 2^m$, and sends a_j to each P_j. We write $[a]_m$ to denote an additive secret sharing of a modulo 2^m. For a prime p, we will abuse notation and use $[a]_p$ to denote a secret sharing of a modulo p.

Revealing a value: To open a value $[a]_m$, every party P_i sends its value a_i to P_{i-1} and P_{i+1}.

Addition by a constant: To add constant c to $[a]_m$, i.e., compute $[b]_m$ with $b = c + a \mod 2^m$, we let P_1 locally compute $b_1 = a_1 + c \mod 2^m$, while P_2 and P_3 just set their $b_i = a_i$.

Addition: Addition of two values $[a]_m$ and $[b]_m$ can be performed locally by each party. To compute $[c]_m$, where $c = a + b \mod 2^m$, every party P_i locally adds its shares, i.e., every party computes $c_i = a_i + b_i \mod 2^m$.

Multiplication using a multiplication triple: Given a secret shared multiplication triple $([x]_m, [y]_m, [z]_m)$ with $z = x \cdot y \mod 2^m$ and two secret shared values $[a]_m$ and $[b]_m$, we compute $[c]_m$ with $c = a \cdot b \mod 2^m$ as follows:

1. Open $e = [x]_m + [a]_m$
2. Open $d = [y]_m + [b]_m$
3. Every party P_i computes $[c]_m = [z]_m + e \cdot [b]_m + d \cdot [a]_m - ed$

3.3 Additive Replicated Secret Sharing

We will use additive replicated secret sharing in our preprocessing protocol in Section 6. Since our preprocessing protocol focuses on the three-party case, we will also restrict our attention to this case here.

Sharing a value: If party P_i wants to share a value $a \in \mathbb{Z}_{2^m}$, it sets $a_i = 0$ and then samples $a_{i+1}, a_{i-1} \leftarrow \mathbb{Z}_{2^m}$ under the constraint that $a = a_1 + a_2 + a_3 \mod 2^m$. It then sends a_{j-1} and a_{j+1} to each P_j.[6] We write $[\![a]\!]_m$ to denote an additive replicated secret sharing of a modulo 2^m. As above, we will abuse notation and for a prime p we will write $[\![a]\!]_p$ to denote the replicated additive secret sharing modulo p.

Sharing a random value: This is a subroutine which will be useful in later protocols. If the parties want to generate shares of a random value they can do it in the following two ways:

Unconditionally secure version: Each party P_i picks a random $s_{i-1} \in \mathbb{Z}_{2^m}$ and sends it to P_{i+1} while at the same time receiving s_{i+1} from P_{i-1}.

Computationally secure version: At the beginning of the protocol, once and for all, the parties run the unconditionally secure version, and interpret their shares as PRF keys K_1, K_2, K_3 such that party P_i knows K_{i-1} and K_{i+1}. When they want to generate the j-th random share, the parties define their shares $s^j_{i-1} = F_{K_{i-1}}(j)$ and $s^j_{i+1} = F_{K_{i+1}}(j)$.

Revealing a shared value: To reveal a secret shared value $[\![a]\!]_m$, each party P_i sends a_{i-1} to P_{i-1} and a_{i+1} to P_{i+1}. Each P_j receives a_j from P_{j-1} and P_{j+1}, checks consistency of the received values, and outputs $a = a_1 + a_2 + a_3 \mod 2^m$ if the check passed. *(Computational Security:)* Opening several values $a^{(1)}, \ldots, a^{(n)}$ can be optimized with computional security as follows: Each P_j only receives $a^{(l)}_j$ from P_{j-1} and computes $a^{(l)} = a^{(l)}_1 + a^{(l)}_2 + a^{(l)}_3 \mod 2^m$. In addition the parties broadcast hashes of $(a^{(1)}, \ldots, a^{(n)})$ and abort in case of a mismatch.

Addition by a constant: To add a public constant c to a secret shared value $[\![a]\!]_m$, i.e., to compute $[\![b]\!]_m$, where $b = c + a \mod 2^m$, we set $b_1 = a_1 + c$, $b_2 = a_2$, and $b_3 = a_3$.

Addition: The addition of two values $[\![a]\!]_m$ and $[\![b]\!]_m$ can be performed locally by each party. To compute $[\![c]\!]_m$, where $c = a + b \mod 2^m$ every party P_i locally adds their shares, i.e., it computes $c_{i-1} = a_{i-1} + b_{i-1} \mod m$ and $c_{i+1} = a_{i+1} + b_{i+1} \mod 2^m$.

Multiplication by a constant: To multiply $[\![a]\!]_m$ by constant c, i.e., to obtain $[\![b]\!]_m$ with $b = c \cdot a \mod 2^m$, every party P_i computes $b_{i-1} = c \cdot a_{i-1} \mod 2^m$ and $b_{i+1} = c \cdot a_{i+1} \mod 2^m$.

Optimistic multiplication: Given $[\![a]\!]_m$ and $[\![b]\!]_m$, we can compute $[\![c]\!]_m$ optimistically, where $c = a \cdot b \mod 2^m$ as follows:

1. The parties generate shares of a random value $[\![s]\!]_m$ as described above.
2. Each P_i computes $u_{i+1} = a_{i+1}b_{i+1} + a_{i+1}b_{i-1} + a_{i-1}b_{i+1} + s_{i-1}$ and sends u_{i+1} to P_{i-1};
3. P_i receives u_{i-1}, thus defining $[\![u]\!]_m$;
4. The parties compute $[\![c]\!]_m = [\![u]\!]_m - [\![s]\!]_m$;

[6] This is a small yet non-trivial optimization of the protocol of [DOS18], where a_i is also a random share and the other two parties have to echo this value to each other to check consistency. By setting $a_i = 0$ we save communication of 4 ring elements per input gate, and one additional round of communication. Note that this change has no impact on security. Remember that we have at most one corrupt party. If P_i is corrupt we need that the two other parties receive the same value of a_i, and this is trivially achieved by setting the value to 0. If one of the other two parties is corrupt, they would learn this share anyway so whether it is random or a constant value it has not impact on security.

3.4 Additive Replicated Secret Sharing with Redundant Shares

In some of our protocols we use a different kind of replicated secret sharing, which we denote as $[\![x]\!]_{m,\lambda}$. Those are sharing of values in $\mathbb{Z}_{2^m}$ but represented with shares in $\mathbb{Z}_{2^{m+\lambda}}$, where λ is a security parameter. Those shares work as the regular additive replicated secret sharings described in the previous sections (e.g., all basic commands are unchanged), but employ shares in a larger ring – this is useful for checking correctness of multiplication triples as we shall see. We describe some basic protocols that can be run with this kind of shares:

Share Coversion (Up): To convert $[\![x]\!]_m$ to $[\![x]\!]_{m,\lambda}$ each party simply interprets their shares as elements of the larger ring (in other words, the shares are padded with 0s in the λ most significant positions). Note that in general $\sum_i x_i \mod 2^{m+\lambda} \neq x$, that is the sum can be either equal to x or to $x + 2^m$ depending on the magnitude of the shares. However, since the semantic of our sharing is that the shared value is $\sum_i x_i \mod 2^m$ the protocol is indeed correct (and this notation allows us a simpler description of more advanced protocols).

Share Conversion (Down): To convert $[\![x]\!]_{m,\lambda}$ to $[\![x]\!]_m$ each party P_i reduces their shares modulo 2^m i.e., compute $x'_{i+1} = x_{i+1} \mod 2^m$ and $x'_{i-1} = x_{i-1} \mod 2^m$.

Both operations preserve the shared value because computing modulo 2^m and modulo $2^{m+\lambda}$ are commutative as 2^m divides $2^{m+\lambda}$ and both operations trivially preserve the replication of shares.

3.5 Additive Replicated Secret Sharing over the Integers

Finally, we recall the replicated secret sharing over integers used by Damgård et al. [DOS18]. The authors observed that one can secret share a value $a \in \mathbb{Z}_{2^m}$ over the integers using shares with bit-length $m + \lambda$. The λ extra bits ensure that the statistical distance between the distributions of shares for any two values in $\mathbb{Z}_{2^m}$ is negligible in λ.

Sharing a value: To share a value $a \in \mathbb{Z}_{2^m}$, P_i picks $a_1, a_2 \leftarrow \{0, \ldots, 2^{m+\lambda} - 1\}$ and sets $a_3 = a - a_1 - a_2$. The shares are distributed among the parties as above. We write $[\![a]\!]_{\mathbb{Z}}$ to denote an additive replicated secret sharing of a over the integers.

Optimistic multiplication: Optimistic multiplication of $[\![a]\!]_{\mathbb{Z}}$ and $[\![b]\!]_{\mathbb{Z}}$ is similar to optimistic multiplication modulo p. Let B be a bound on the share amplitude. To optimistically compute $[\![c]\!]_{\mathbb{Z}}$ with $c = a \cdot b$ we do the following:

1. The parties generate shares of a random value $[\![s]\!]_{\mathbb{Z}}$ as described above, but where s_i are chosen in $\{0, \ldots, 2^{2\lceil \log B \rceil + \lambda + 2} - 1\}$ (if the information-theoretic version of the generation of s_i is used, parties also check that the received shares are in the right range);
2. Each P_i computes u_{i+1} as described above (but over the integer);
3. P_i receives u_{i-1} and checks that $|u_{i-1}| \leq 2^{2\lceil \log B \rceil + \lambda + 3}$;
4. The parties compute $[\![c]\!]_m = [\![u]\!]_m - [\![s]\!]_m$;

All other operations, are analogous to their counterparts modulo m. For a more detailed exposition refer to [DOS18].

4 Extension of the Compiler by Damgård et al.

The compiler $\mathsf{COMP}_{\mathsf{old}}$ by Damgård et al. [DOS18] takes an n-party passively $(t^2 + t)$-secure protocol Π and transforms it into a protocol $\mathsf{COMP}_{\mathsf{old}}(\Pi)$ that is secure with abort against t active corruptions[7]. For example, for $t = 1$, the compiler can transform a passively two-secure three-party protocol into a protocol that is secure against one active corruption. The high-level idea of the compiler is to let virtual parties

[7] The authors also show how to achieve active security with guaranteed output delivery, but here we only focus on the case of security with abort.

execute the passively secure protocol on behalf of the real parties. Each virtual party $\mathbb{P}_i$ is simulated by $t+1$ real parties $\mathrm{P}_i, \dots, \mathrm{P}_{i+t}$ in a way that prevents an active adversary, who controls at most t real parties, from actively corrupting any of the virtual parties. In the following we will write $\mathrm{P}_j \in \mathbb{P}_i$ to denote that real party P_j is simulating virtual party $\mathbb{P}_i$.

The workflow of their compiler can be split into two phases. In the first phase, for each virtual party $\mathbb{P}_i$, all real parties $\mathrm{P}_j \in \mathbb{P}_i$ agree on a common input and randomness that will be used by $\mathbb{P}_i$ during the execution of the passively secure protocol Π. Having the same input and the same randomness, every $\mathrm{P}_j \in \mathbb{P}_i$ will be able to redundantly compute the exact same messages that $\mathbb{P}_i$ is supposed to send during the execution of Π. In the second phase, the virtual parties run Π to compute the desired functionality from the inputs and randomness that the virtual parties have agreed upon. Whenever $\mathbb{P}_i$ is supposed to send a message to $\mathbb{P}_j$ according to Π, *every* real party simulating $\mathbb{P}_i$ will send a separate message to *every* real party simulating $\mathbb{P}_j$. Each real party verifies that it receives the same message from all sending real parties and aborts if this is not the case.

Intuitively, the resulting protocol is secure against t active corruptions, since an adversary cannot misbehave on behalf of a virtual party it is simulating, and at the same time be consistent with at least one other honest real party that simulates the same virtual party.

From an efficiency point of view, every message from one $\mathbb{P}_i$ to some other $\mathbb{P}_j$ is sent redundantly from $t+1$ to $t+1$ real parties. That is, if the passively secure protocol Π sends ℓ messages during a protocol execution, then $\mathsf{COMP}_{\mathsf{old}}(\Pi)$ will send roughly $\mathcal{O}\left(\ell \cdot t^2\right)$ many messages.

4.1 A New Compiler for Protocols with Weak Privacy

We present a new compiler $\mathsf{COMP}_{\mathsf{new}}$, which makes slightly stronger assumptions about the starting protocol Π, but compiles it into an actively secure protocol in a more communication efficient manner. $\mathsf{COMP}_{\mathsf{new}}$ takes as input a $\left(t^2+t\right)$-weakly private protocol Π and outputs a compiled protocol $\mathsf{COMP}_{\mathsf{new}}(\Pi)$ that is secure against t active corruptions. If Π sends ℓ messages in total, then our compiled protocol will only send $\mathcal{O}\left(\ell \cdot t + t^2\right)$ messages.

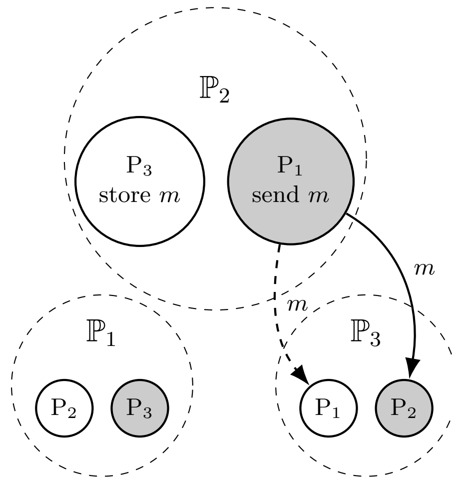


Fig. 3: An illustration of our simulation strategy for the case of three parties with one active corruption. Dashed circles represent virtual parties. Solid circles inside the dashed circles represent the real parties that simulate the given virtual party. The brains of each virtual party are highlighted in gray. The figure illustrates the process of virtual party $\mathbb{P}_2$ sending a message to virtual party $\mathbb{P}_3$. The black arrows indicate that P_1, the brain of $\mathbb{P}_2$, sends one message to P_2 and one to P_1, which is omitted in reality, since it is sending a message to itself. P_3 stores this message in its transcript.

Our new compiler follows the approach of $\mathsf{COMP}_{\mathsf{old}}$. However, instead of verifying the validity of every single message between virtual parties as soon as it is sent, we will let the real parties simulate the virtual parties in a more optimistic and communication efficient fashion, where the correctness of all communicated messages is only verified once at the end of the computation phase, right before the opening phase of Π. Pushing the whole verification to the end of the computation phase allows us to reduce the total number or redundant messages that are sent. This new simulation strategy crucially relies on the weak active privacy of Π, since we are now allowing the adversary to misbehave up to the opening phase without aborting the protocol execution.

The first phase of $\mathsf{COMP}_{\mathsf{new}}$, where all parties agree on their inputs and random tapes, is identical to that of $\mathsf{COMP}_{\mathsf{old}}$ and is thus equally efficient. In the second phase, our new simulation approach works by selecting one arbitrary real party P_i in each virtual party $\mathbb{P}_j$ to be the brain $\mathrm{B}_j := \mathrm{P}_i$ of that virtual party. The brains will act on behalf of their corresponding virtual parties in an optimistic fashion and execute the computation phase of Π up to the opening phase. All other real parties, the pinkies, will receive the messages that their corresponding virtual parties should receive, which enables them to follow the protocol locally. However, the pinkies will not send any messages during the computation phase. They will only become actively involved in the opening phase to ensure that all brains behaved honestly during the computation phase. Once correctness is ensured, all parties will jointly perform the opening phase of Π. During the computation phase of Π, whenever virtual party $\mathbb{P}_i$ is supposed to send a message to virtual party $\mathbb{P}_j$, we let B_i send one message to each real party in $\mathbb{P}_j$. The receiving real parties do not perform any checks at this moment and just store the message. B_j will optimistically continue the protocol execution on behalf of $\mathbb{P}_j$ according to Π and the received message. This simulation strategy is illustrated in Figure 3.

At the end of the computation phase, all real parties jointly make sure that for each pair $(\mathbb{P}_i, \mathbb{P}_j)$, the sending virtual party $\mathbb{P}_i$ always behaved honestly towards the receiving virtual party $\mathbb{P}_j$. This is accomplished by using a message checking protocol (that implements $\mathcal{F}_{\mathsf{check}}$). If any of these checks output $\mathsf{different}$, then the protocol execution is aborted.

In the opening phase, after passing the previous check, every virtual party is supposed to send its last opening message to all other virtual parties. For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, all real parties in $\mathbb{P}_i$ send the last message to all real parties in $\mathbb{P}_j$. Every receiving party checks that all $t+1$ received messages are consistent and aborts if this is not the case.

In our formal description, let $f(x_1, \ldots, x_n)$ be the n-party functionality that we want to compute. For the sake of simplicity and without loss of generality, we assume that all parties learn the output of the computation. Let $\mathbb{P}_i$ be the virtual party that is simulated by real parties $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$. Let $\mathbb{V}_i$ be the set of virtual parties in whose simulation P_i participates. Let f' be a related n-party functionality that takes as input $\left(x_1^i, \ldots, x_n^i\right)$ from every P_i and outputs $f(\sum_{i=1}^n x_1^i, \ldots, \sum_{i=1}^n x_n^i)$. That is, every party inputs one secret share of every original input. The functionality f' reconstructs the original inputs for f from the secret shares and then evaluates f on those inputs. Let $\Pi_{f'}$ be a passively $\left(t^2+t\right)$-secure protocol with robust privacy that securely implements $\mathcal{F}_{f'}$. The formal description of our compiler is given in Figure 4. Throughout our description we assume that honest parties consider message that they do not receive as malicious and act accordingly.

Theorem 1. *Let $n \geq 3$. Assume $\Pi_{f'}$ implements n-party functionality $\mathcal{F}_{f'}$ with $\left(t^2+t\right)$-weak privacy. Then, $\mathsf{COMP}_{\mathsf{new}}\left(\Pi_{f'}\right)$ implements functionality $\mathcal{F}_f$ with active security under individual abort against t corruptions. If $\Pi_{f'}$ has a total bandwidth cost of ℓ messages, then $\mathsf{COMP}_{\mathsf{new}}\left(\Pi_{f'}\right)$ has a total bandwidth cost of $\mathcal{O}\left(\ell \cdot t + t^2\right)$ messages.*

Remark. Similar to Damgård et al. [DOS18], we prove our result for the case of active security with individual abort, where some honest parties may terminate, while some may not. As in their work, our result easily extends to unanimous abort with one additional round of secure broadcast.

Proof. Our proof closely follows the proof of Damgård et al. [DOS18] for the $\mathsf{COMP}_{\mathsf{old}}$ compiler. Let $\mathbb{P}^*$ be the set of corrupted real parties and let $\mathbb{V}^*$ be the set of virtual parties that are simulated by at least one corrupt real party. Let $\mathcal{S}_{f'}$ be the simulator of the (t^2+t)-weakly private protocol $\Pi_{f'}$. We will use this

$\mathsf{COMP}_{\mathsf{new}}\,(\Pi_{f'})$

Inputs: Each party P_i has input x_i.

---

1. **Input sharing:**
   (a) Each P_i secret shares its input $x_i = x_i^1 + \cdots + x_n^i$.
   (b) For $1 \leq j \leq n$, each P_i sends $\left(x_i^j, \mathbb{P}_j\right)$ to the broadcast functionality $\mathcal{F}_{\mathsf{bcast}}$.
   (c) Each P_i receives $z_j := \left(x_1^j, \ldots, x_n^j\right)$ for each $\mathbb{P}_j \in \mathbb{V}_i$ from the broadcast functionality and aborts if any of the shares equals $\perp$.
2. **Randomness:** Each brain B_i chooses a uniformly random string r_i and sends $(r_i, \mathbb{P}_i)$ to $\mathcal{F}_{\mathsf{bcast}}$. The receiving real parties abort if they receive $\perp$.
3. **Computation phase:** All virtual parties jointly execute the computation phase of $\Pi_{f'}$, where each $\mathbb{P}_i$ uses input z_j and random tape r_i, as follows:
   – Whenever $\mathbb{P}_i$ is supposed to send message m to $\mathbb{P}_j$, the brain B_i sends m to all real players in $\mathbb{P}_j$.
   – Whenever $\mathbb{P}_j$ receives message m, all pinkies store the message and only B_j continues the protocol according to $\Pi_{f'}$. The pinkies locally follow the protocol and compute the message that they would send.
4. **Check:** At the end of the computation phase, all parties, brains and pinkies, jointly check that the current transcript is correct. For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, for each party $\mathrm{P}_k \in \mathbb{P}_j$, we invoke $\mathcal{F}_{\mathsf{check}}$, where P_k acts as the receiver and $\mathbb{P}_i$ act as the remaining parties. The input of P_k is the list of messages it received from $\mathbb{P}_i$ and the input of all parties from $\mathbb{P}_i$ is the list of messages that they would have sent. If any invocation outputs $\mathsf{different}$, then the protocol execution is aborted.
5. **Opening phase:**
   (a) For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, all real parties in $\mathbb{P}_i$ send the last message of $\Pi_{f'}$ to all real parties in $\mathbb{P}_j$.
   (b) Every real party in $\mathbb{P}_j$ checks that all received messages are equal. If they are it obtains the output of the computation and otherwise it aborts.

Fig. 4: Formal description of our compiler.

simulator to construct a simulator $\mathcal{S}$ for the overall actively secure protocol $\mathsf{COMP}_{\mathsf{new}}(\Pi_{f'})$. The simulator $\mathcal{S}$ works as follows:

1. For each party $\mathrm{P}_i \in \mathbb{P}^*$ and $j \in [n]$, the adversary $\mathcal{Z}$ sends $(x_i^j, \mathbb{P}_j)$ to the ideal functionality $\mathcal{F}_{\mathsf{bcast}}$, which is emulated by the simulator $\mathcal{S}$. For any invocation that involves a corrupted party, the environment decides which outputs are $\perp$ and which get delivered. For each $\mathbb{P}_j \in \mathbb{V}^*$ and each corrupt real party in $\mathbb{P}_j$, we send back $(x_1^j, \ldots, x_n^j)$, where x_i^j is either the share that was sent by $\mathcal{Z}$ if P_i is corrupt or otherwise a uniformly random share.
2. For each corrupted party $\mathrm{P}_i \in \mathbb{P}^*$, we reconstruct its input as $x_i = \sum_{j=1}^n x_i^j$.
3. $\mathcal{S}$ sends the inputs of the corrupted parties to $\mathcal{F}_f$ and receives back the output of the computation $z = f(x_1, \ldots, x_n)$.
4. For each $\mathbb{P}_i \in \mathbb{V}^*$ we consider two cases. If the brain B_i is corrupted, then it chooses a random tape r_i and sends it to $\mathcal{F}_{\mathsf{bcast}}$, which again is simulated by $\mathcal{S}$. If B_i is honest, then the simulator picks a uniformly random r_i and sends it back to $\mathcal{Z}$ on behalf of $\mathcal{F}_{\mathsf{bcast}}$. Again, the environment can decide that some of the outputs in this step will be $\perp$, which will then be handled accordingly by our simulator.
5. At this point, we know the inputs and the random tapes of all virtual parties $\mathbb{P}_i \in \mathbb{V}^*$. We can therefore compute the exact messages that we would expect from an honest party following the protocol. We initialize the simulator $\mathcal{S}_{f'}$ with parties $\mathbb{P}_1 \ldots \mathbb{P}_n$ and the set of corrupted players $\mathbb{V}^*$.
6. When $\mathcal{S}_{f'}$ queries $\mathcal{F}_{f'}$ for the inputs of the corrupted parties, we provide it with $(x_1^i, \ldots, x_n^i)$ for each $\mathbb{P}_i \in \mathbb{V}^*$.
7. We now describe how to simulate the computation phase of the protocol.

- $\mathcal{S}$ queries $\mathcal{S}_{f'}$ for the messages that the honest brains send to the corrupted virtual parties. For each message m to some $\mathbb{P}_i \in \mathbb{V}^*$, we send m to each corrupted real party in $\mathbb{P}_i$ (unless the sender received $\perp$ in one of step 1 or 4 of this simulator in which case it sends nothing).
- $\mathcal{Z}$ outputs the messages that the corrupt parties send to the honest ones. Since we know the input and random tape of each corrupted party, we can see which messages are honestly generated and which are not. Forward the message of the sending brain to $\mathcal{S}_{f'}$ as the message of $\mathbb{P}_i$.

8. At the end of the computation phase, we simulate the check protocol as follows. For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, for each real party $R \in \mathbb{P}_j$, we have one invocation of the functionality $\mathcal{F}_{\mathsf{check}}$. The simulator $\mathcal{S}$ needs to simulate the ideal functionality towards the corrupted parties in each invocation that involves a corrupted party. Note that the inputs of all honest parties to each check are known from the previous part of the simulation. We, at this point, also know whether any of the corrupted brains cheated or not and if so which check invocation should fail. Furthermore, whenever a corrupted party sends a value to the check functionality, we know whether it's the correct one or not. Using the above observations it follows that the simulator always knows how to simulate each invocation of $\mathcal{F}_{\mathsf{check}}$ and when to return different, when to return same, and when to abort the computation.
9. If all checks passed, meaning that the adversary did not misbehave at any point in time, then we continue the simulation. The simulator $\mathcal{S}$ knows all the last messages of each corrupted party and it knows the output of the functionality z. Since the opening phase is a linear reconstruction of the last messages, the simulator picks a uniformly last message for each honest party under the condition that the linear combination of all last messages results in z. The simulator faithfully executes the last step of the protocol compiler with the corrupted parties. For any simulated honest real party that receives incorrect messages from $\mathcal{Z}$, we will instruct $\mathcal{F}_f$ to make this party abort. For any honest real party that receives the correct last round messages, we instruct $\mathcal{F}_f$ to deliver the output of the computation.

The simulation of the first protocol phase (steps 1-4) is perfect. The adversary sees uniformly random shares, random tapes, or of the things it sent itself just like in a real protocol execution. The simulation of $\mathcal{F}_{\mathsf{bcast}}$ is identical to a real execution. The indistinguishability of the simulation in step 6 directly follows from the security guarantees of $\mathcal{S}_{f'}$. As in the real execution, we do not send anything from real honest parties that may have aborted during the first phase. Otherwise, in both the real and the ideal world, the protocol does not abort during the computation phase. During the computation phase the simulator has access to the random tapes and inputs of the corrupted parties, thus always knows when and where cheating occurred. This enables us to correctly determine when and where the protocol would abort and simulate the outcome of the check phase in step 8 correctly.

5 Efficient Three-Party Computation

All of our protocols are fundamentally based on the seminal work of Beaver [Bea92], who presented a conceptually simple and clean approach for passively secure circuit evaluation. We present two flavors of protocols. One with preprocessing and two different instantiations of the preprocessing phase and one without preprocessing, but some light postprocessing. The protocols that involve preprocessing have a larger *total* runtime, but a leaner online phase, whereas the postprocessing protocol has a smaller overall runtime. Our protocol with preprocessing is presented in this section and the two different preprocessing protocols are presented in Section 6 Our protocol with postprocessing is presented in Section 7.

5.1 Beaver's Circuit Evaluation Approach

The circuit evaluation approach by Beaver [Bea92] enables, in our case, three parties to evaluate an arithmetic circuit f over arbitrary rings $\mathbb{Z}_{2^m}$ with security against two passive corruptions. The protocol is split into a preprocessing and an online phase. During the preprocessing phase the parties jointly generate some function-independent correlated randomness in the form of additively secret shared multiplication triples $[a_i]_m$, $[b_i]_m$, $[c_i]_m$, where $c_i = a_i \cdot b_i \mod 2^m$. In the online phase these triples are then consumed to securely evaluate

some desired function f. Beaver's online phase works in three steps. First, all parties additively secret share their input among the other parties. Then, all parties jointly evaluate the circuit in a gate-by-gate fashion on the secret shared values. Additions are performed locally, and multiplications require interaction as well as correlated randomness as explained in Section 3.2. In the last step, the parties jointly reconstruct the secret shared values of the output wires of the circuit. Note that the reconstruction phase is just a linear function of the messages received during the opening phase.

Proposition 1. *Let f be an arithmetic circuit with N multiplication gates. Given N preprocessed multiplication triples, the three-party protocol Beaver_f, implements functionality $\mathcal{F}_f$ with 2-weak privacy and has linear reconstruction.*

5.2 Our Protocol with Preprocessing

We focus on the popular setting with three parties and one active corruption and obtain our protocol by applying Theorem 1 to Beaver's circuit evaluation approach. Let f be the three-party functionality that shall be computed, where each party P_i has an input $x_i \in \mathbb{Z}_{2^m}$. As before, let f' be the related three-party functionality that first recomputes the original inputs from the additive secret shares and then evaluates f. Let N be the number of multiplication gates in f and assume for the moment that all real parties have already shared this many *replicated* secret shares of multiplication triples $[\![a_i]\!]_m$, $[\![b_i]\!]_m$, $[\![c_i]\!]_m$ in a preprocessing phase[8]. Our concrete preprocessing protocol will be described in detail in Section 6. Since for $1 \leq i \leq 3$, virtual party $\mathbb{P}_i$ will be simulated by P_{i-1} and P_{i+1}, it holds that real parties holding replicated shares is equivalent to virtual parties holding additive secret shares. This way, one can think of those replicated shares as parts of the real parties' inputs that have already been shared correctly among the virtual parties during preprocessing. We state the compiled protocol $\mathsf{COMP}_{\mathsf{new}}(\mathsf{Beaver}_{f'})$ in Figure 5.

Concrete Efficiency. We explicitly state the communication complexity of the protocol: Addition gates require no communication. Evaluating a multiplication gate requires sending 6 words of m bit each. The opening phase, including the checking protocol, requires sending 5 hash values (we choose 256 as the output of the hash) as well as the output shares giving a total of $1280 + 4m \cdot |\mathrm{out}|$ bits for out output gates.

6 Preprocessing

During the preprocessing phase we generate replicated secret sharings of multiplication triples $c = a \cdot b \mod 2^m$. We describe two versions of the preprocessing phase.

The first is obtained combining the preprocessing of Damgård et al. [DOS18] with the batch verification technique of Ben-Sasson et al. [BFO12]. The generation of multiplication triples is split in three steps. First, using the techniques of Damgård et al., we optimistically generate secret shared multiplication triples over the integers. Next, we interpret them as triples in a field $\mathbb{Z}_p$, for some sufficiently large prime p, and perform the batch verification protocol of Ben-Sasson et al. to ensure that all triples are correct. Lastly, we reduce all integer shares modulo 2^m to obtains shares of multiplication triples in our desired ring $\mathbb{Z}_{2^m}$[9].

The second version of the protocol is inspired by the work of Cramer et al. [CDE+18], where MACs modulo some prime are replaced with MACs modulo $2^{m+\lambda}$ with 2^m being the "plaintext space" and λ being a statistical security parameter. Following this approach, we replace the computation modulo prime in the correctness check with a check performed over $2^{m+\lambda}$, which still guarantees security. Due to the computational complexity of performing computation modulo primes, this second version of the preprocessing offers significantly improved performances.

[8] The functionalities f and f' have equally many multiplication gates, since reconstructing the inputs from additive secret shares does not require any multiplications.

[9] Valid multiplication triples over integers are valid multiplication triples modulo 2^m.

$\mathsf{COMP}_{\mathsf{new}}\,(\mathsf{Beaver}_{f'})$

Inputs: Each party P_i has input $x_i \in \mathbb{Z}_{2^m}$ and they all share preprocessed triples $[\![a_j]\!]_m, [\![b_j]\!]_m, [\![c_j]\!]_m$ for $j \in \{1, \ldots, N\}$.

---

1. **Input sharing:**
   (a) Each P_i picks $x_i^1, x_i^2 \leftarrow \mathbb{Z}_{2^m}$ and sets $x_i^3 = x_i - x_i^1 - x_i^2 \mod 2^m$.
   (b) For $1 \leq j \leq 3$, each P_i sends $\left(x_i^j, \mathbb{P}_j\right)$ to the broadcast functionality $\mathcal{F}_{\mathsf{bcast}}$.
   (c) Each P_i receives $z_{i-1} := \left(x_1^{i-1}, x_2^{i-1}, x_3^{i-1}\right)$ and $z_{i+1} := \left(x_1^{i+1}, x_2^{i+1}, x_3^{i+1}\right)$ via the broadcast and aborts if any of the shares equals $\perp$.
2. **Randomness:** Each brain B_i chooses a uniformly random string r_i and sends $(r_i, \mathbb{P}_i)$ to $\mathcal{F}_{\mathsf{bcast}}$. The receiving real parties abort if they receive $\perp$.
3. **Computation phase:** All virtual parties now evaluate $\mathsf{Beaver}_{f'}$ in a gate-by-gate fashion, where each $\mathbb{P}_i$ uses input z_i as follows:
   – Multiplication gates are evaluated using correlated randomness.
   – All other types of gates are executed locally.
4. **Check:** For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, we use $\mathcal{F}_{\mathsf{check}}$ to verify the correctness of the messages sent from $\mathbb{P}_i$ to $\mathbb{P}_j$.
5. **Opening phase:**
   (a) For each output wire w, for each pair $(\mathbb{P}_i, \mathbb{P}_j)$, all real parties in $\mathbb{P}_i$ send their secret share of w to all real parties in $\mathbb{P}_j$.
   (b) For all $\mathbb{P}_j$, all $\mathrm{P}_k \in \mathbb{P}_j$ check that all messages they received are equal. If not, abort.
   (c) If all received shares are consistent, then reconstruct the output wire value w and terminate.

Fig. 5: Protocol for three-party arithmetic circuit evaluation over the ring $\mathbb{Z}_{2^m}$ with active security with abort against one active corruption.

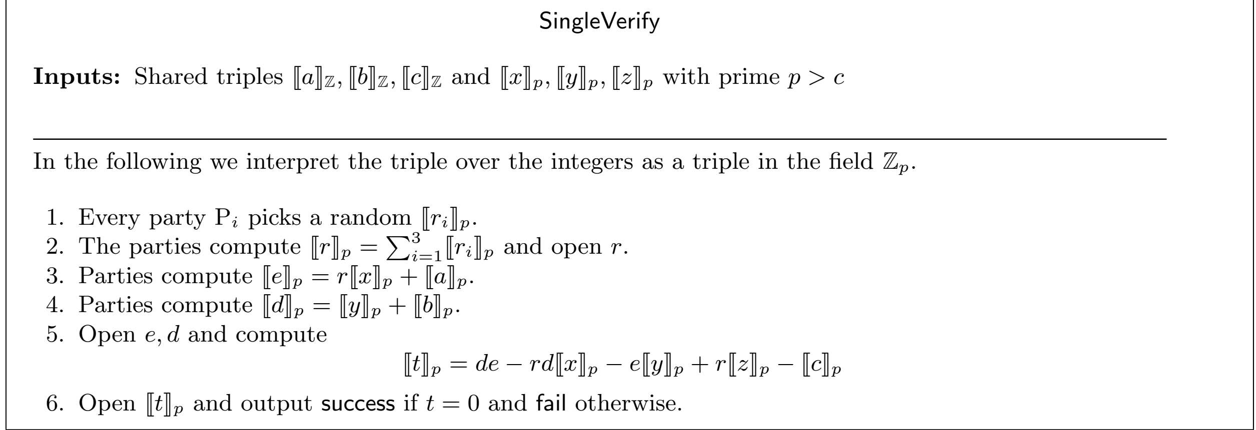
6.1 [DOS18]-style Preprocessing

Optimistic Generation of Multiplication Triples Optimistic generation of a multiplication triple over the integers is straightforward. First each party P_i uses replicated secret sharing over the integers to share random values $a_i, b_i \in \mathbb{Z}_{2^m}$. All parties jointly compute $[\![a]\!]_{\mathbb{Z}} = \sum_{i=1}^{3} [\![a_i]\!]_{\mathbb{Z}}$ and $[\![b]\!]_{\mathbb{Z}} = \sum_{i=1}^{3} [\![b_i]\!]_{\mathbb{Z}}$ and then use the optimistic multiplication of replicated secret shares from Section 3.5 to compute $[\![c]\!]_{\mathbb{Z}}$.

Verification of a Single Multiplication Triple Given an optimistically generated triple $[\![a]\!]_{\mathbb{Z}}, [\![b]\!]_{\mathbb{Z}}, [\![c]\!]_{\mathbb{Z}}$, the preprocessing of Damgård et al. proceeds as follows. First, the authors optimistically generate another multiplication triple in $\mathbb{Z}_p$, where p is a prime such that $p > c$. Then they interpret the multiplication triple over the integers as a triple in $\mathbb{Z}_p$ and employ the standard technique of "sacrificing" one triple to check the other one [DO10]. Concretely, the authors sacrifice the triple in $\mathbb{Z}_p$ to check the triple over the integers. The check, SingleVerify, is described in detail in Figure 6. The rationale behind this approach is that if the multiplicative relation $a \cdot b = c$ holds over the integers, then it also holds in $\mathbb{Z}_p$, since $p > c$ and thus no wrap-around due to the modulo operation happens.

Efficient Batch Verification Now given N optimistically generated multiplication triples $[\![a_i]\!]_{\mathbb{Z}}$, $[\![b_i]\!]_{\mathbb{Z}}$, $[\![c_i]\!]_{\mathbb{Z}}$ over the integers, we would like to efficiently check that, for all $i \in \{1, \ldots N\}$, the multiplicative relationship $a_i \cdot b_i = c_i$ holds. Checking every multiplication triple separately, would require us to generate N additional multiplication triples in $\mathbb{Z}_p$ and perform N invocations of SingleVerify.

Instead, we use a clever idea of Ben-Sasson et al. [BFO12]. Using their technique for verifying the validity of multiplication triples allows us to verify N triples with N additional optimistic multiplications and a



SingleVerify

Inputs: Shared triples $[\![a]\!]_{\mathbb{Z}}, [\![b]\!]_{\mathbb{Z}}, [\![c]\!]_{\mathbb{Z}}$ and $[\![x]\!]_p, [\![y]\!]_p, [\![z]\!]_p$ with prime $p > c$

In the following we interpret the triple over the integers as a triple in the field $\mathbb{Z}_p$.

1. Every party P_i picks a random $[\![r_i]\!]_p$.
2. The parties compute $[\![r]\!]_p = \sum_{i=1}^{3} [\![r_i]\!]_p$ and open r.
3. Parties compute $[\![e]\!]_p = r[\![x]\!]_p + [\![a]\!]_p$.
4. Parties compute $[\![d]\!]_p = [\![y]\!]_p + [\![b]\!]_p$.
5. Open e, d and compute
$$[\![t]\!]_p = de - rd[\![x]\!]_p - e[\![y]\!]_p + r[\![z]\!]_p - [\![c]\!]_p$$
6. Open $[\![t]\!]_p$ and output success if $t = 0$ and fail otherwise.

Fig. 6: Verification of multiplication triple $([\![a]\!]_{\mathbb{Z}}, [\![b]\!]_{\mathbb{Z}}, [\![c]\!]_{\mathbb{Z}})$. SingleVerify takes as input two potentially incorrect multiplication triples and sacrifices one to check the other.

single invocation of SingleVerify. The main idea behind their approach is to encode all multiplication triples $(a_1, b_1, c_1), \ldots, (a_N, b_N, c_N)$ as three polynomials (f, g, h), where the relation $f \cdot g = h$ will hold iff all multiplication triples are correct. Instead of checking each multiplication triple separately, we will evaluate the polynomials at a random point z and and verify that the polynomial relation $f(z) \cdot g(z) \equiv h(z) \mod p$ holds.

More concretely, let f and g be polynomials with coefficients in $\mathbb{Z}_p$ of degree N-1 with $f(i) = a_i$ and $g(i) = b_i$. These polynomials are uniquely defined by the values $a_1, \ldots, a_N$ and $b_1, \ldots, b_N$. Since, we expect h to be $f \cdot g$ and thus of degree $2N - 2$, we require $2N - 1$ points to uniquely define it. For $i \in \{1, \ldots N\}$, we set $h(i) = c_i$. For $i \in \{N+1, \ldots 2N-1\}$, we set $h(i) = f(i) \cdot g(i)$, where the multiplication of $f(i)$ and $g(i)$ is performed optimistically. If all multiplication triples and all optimistic multiplications are correct, then $f \cdot g = h$ holds and an evaluation at a random point z will always fulfil $f(z) \cdot g(z) \equiv h(z) \mod p$. If, however, some multiplication triple is not valid, then $f \cdot g \neq h$ and in this case the two polynomials $f \cdot g$ and h can agree on at most $2N - 2$ many points. This means that for a uniformly random point $z \in \mathbb{Z}_p$, we have $\Pr[f(z) \cdot g(z) = h(z) \mid f \cdot g \neq h] \leq \frac{2N-2}{|\mathbb{Z}_p|}$.

This trick crucially relies on the fact that we can interpolate and evaluate additively secret shared polynomials. Say we are given shares of points $[\![a_i]\!]_p$ for $i \in \{1, \ldots, N\}$ of polynomial f and we would like to evaluate f on point z. Define Kronecker delta $\delta_i^N(x)$ as

$$\delta_i^N(x) := \prod_{j=1, j\neq i}^{N} \frac{x-j}{i-j} = \begin{cases} 1 & x = i \\ 0 & x \neq i \end{cases}$$

Evaluating polynomial f at point z is then done by computing

$$[\![f(z)]\!]_p = \sum_{i=1}^{N} \left(\delta_i^N(z) \cdot [\![a_i]\!]_p\right)$$

We provide a formal description of the batch verification protocol in Figure 7. Its security directly follows from the security of the preprocessing of Damgård et al. and the batch verification protocol of Ben-Sasson et al. Let Π_{Triple} be the protocol that first optimistically generates N multiplication triples over the integers, then executes the batch verification, and finally reduces all shares modulo 2^m. The concrete efficiency of this protocol is discussed in Section 8.

BatchVerify

Inputs: Parties P_1, P_2, and P_3 share N preprocessed triples $[\![a_i]\!]_{\mathbb{Z}}, [\![b_i]\!]_{\mathbb{Z}}, [\![c_i]\!]_{\mathbb{Z}}$ for $i \in \{1, \ldots, N\}$ over the integers. Let $[\![x]\!]_p, [\![y]\!]_p, [\![z]\!]_p$ be a uniformly random triple from $\mathbb{Z}_p$.

In the following we interpret each triple over the integers as a triple in the field $\mathbb{Z}_p$, where p is a sufficiently large prime.

1. For $i \in \{1, \ldots, N\}$, define $[\![f(i)]\!]_p := [\![a_i]\!]_p$ and $[\![g(i)]\!]_p := [\![b_i]\!]_p$.
2. For $i \in \{N+1, \ldots, 2N-1\}$, compute

$$[\![f(i)]\!]_p := \sum_{j=1}^{N} \left(\delta_j^N(i) \cdot [\![a_j]\!]_p \right), \text{ and}$$

$$[\![g(i)]\!]_p := \sum_{j=1}^{N} \left(\delta_j^N(i) \cdot [\![b_j]\!]_p \right)$$

3. For $i \in \{1, \ldots, N\}$, define $[\![h(i)]\!]_p := [\![c_i]\!]_p$.
4. For $i \in \{N+1, \ldots, 2N-1\}$, compute $[\![h(i)]\!]_p = [\![f(i)]\!]_p \cdot [\![g(i)]\!]_p$ optimistically.
5. Every party P_i picks a random $[\![z_i]\!]_p$.
6. The parties compute $[\![z]\!]_p = \sum_{i=1}^{3} [\![z_i]\!]_p$ and open z.
7. Compute:

$$[\![\alpha]\!]_p = [\![f(z)]\!]_p := \sum_{j=1}^{2N-1} \left(\delta_j^N(z) \cdot [\![f(j)]\!]_p \right), \text{ and}$$

$$[\![\beta]\!]_p = [\![g(z)]\!]_p := \sum_{j=1}^{2N-1} \left(\delta_j^N(z) \cdot [\![g(j)]\!]_p \right), \text{ and}$$

$$[\![\gamma]\!]_p = [\![h(z)]\!]_p := \sum_{j=1}^{2N-1} \left(\delta_j^N(z) \cdot [\![h(j)]\!]_p \right)$$

8. Check SingleVerify $([\![\alpha]\!]_p, [\![\beta]\!]_p, [\![\gamma]\!]_p, [\![x]\!]_p, [\![y]\!]_p, [\![z]\!]_p)$.

Fig. 7: Batch verification of multiplication triples.

6.2 [CDE+18]-style Preprocessing

Optimistic Generation of Multiplication Triple: As in the previous subsection, optimistic generation of a multiplication triple is straightforward. This time, the parties (using replicated secret sharing), generate random sharings $[\![x]\!]_m$, $[\![y]\!]_m$ and then use the optimistic multiplication protocol to compute $[\![z]\!]_m$.

Verification of a Single Multiplication Triple: We present here a verified multiplication protocol in $\mathbb{Z}_{2^m}$ where, in order to mitigate zero divisors, most of the computation is executed in $\mathbb{Z}_{2^{m+\lambda}}$ for a statistical parameter λ. The techniques used in this kind of preprocessing are inspired by the protocol SPD$\mathbb{Z}_{2^k}$ [CDE+18]. However, here they are used in a very different context, since SPD$\mathbb{Z}_{2^k}$is a protocol for the dishonest majority case (and therefore their preprocessing phase requires expensive public-key operations), while our protocol can be instantiated using only cheap field operations (as we are working in the honest majority case).

Figure 8 presents the basic version of our protocol (the notation $[\![\cdot]\!]_{m,\lambda}$ used in this section was introduced in Section 3.4). At its core, the protocol uses the sacrifice step of the MASCOT protocol [KOS16]. (The protocol is here used for checking correctness of preprocessed multiplication triples, but in the next section we will re-use this protocol introduce the protocol in the context of our postprocessing based protocol.).

SPD$\mathbb{Z}_{2^k}$-like check for correct multiplication

Input $[\![x]\!]_m$, $[\![y]\!]_m$, $[\![z]\!]_m$
Output A flag f indicating whether $z = xy \mod 2^m$ or not

---

1. The parties convert the input shares $([\![x]\!]_m, [\![y]\!]_m, [\![z]\!]_m)$ into $([\![x]\!]_{m,\lambda}, [\![y]\!]_{m,\lambda}, [\![z]\!]_{m,\lambda})$.
2. The parties generate a random $[\![a]\!]_{m,\lambda}$ and execute an optimistic multiplication with $([\![a]\!]_{m,\lambda}, [\![y]\!]_{m,\lambda})$ to get $[\![c]\!]_{m,\lambda}$.
3. The parties jointly generate a random $r \in \mathbb{Z}_{2^\lambda}$.
4. The parties reveal $[\![e]\!]_{m,\lambda} = r[\![x]\!]_{m,\lambda} + [\![a]\!]_{m,\lambda}$.
5. The parties output the result of the equality check

$$r[\![z]\!]_{m,\lambda} + [\![c]\!]_{m,\lambda} - e[\![y]\!]_{m,\lambda} \stackrel{?}{=} 0 \tag{1}$$

Fig. 8: The protocol to check correctness of a multiplication triple using redundant $[\![\cdot]\!]_{m,\lambda}$ sharing.

Note that generating the share of a can be done non-interactively using PRSS. Similarly, r can be generated using PRSS and then opening the result. We now evaluate the properties of our protocol.

Correctness. Correctness follows easily thanks to the correctness of optimistic multiplication and that the fact that

$$rz + c - ey = rxy + ay - (rx + a) \cdot y = 0 \mod 2^{m+\lambda}.$$

Security. Assuming that all openings are verified using $\mathcal{F}_{\mathsf{check}}$ (which ensures that a corrupt party cannot send different shares to different parties), the corrupt party can only deviate by adding an additive error in the optimistic products. We define the (potential) errors as $z = xy + \epsilon_z$ and $c = ay + \epsilon_c$. If the input tuple $[\![x]\!]_m, [\![y]\!]_m, [\![z]\!]_m$ is incorrect then we have that $\epsilon_z \neq 0 \mod 2^m$. Inserting into the check equation (1), we get

$$rxy + r\epsilon_z + ay + \epsilon_c - (rx + a)y = r\epsilon_z + \epsilon_c \mod 2^{m+\lambda}.$$

Since r is a uniformly random λ-bit number, it follows that $r\epsilon_z \bmod 2^{m+\lambda}$ is uniform in a set of at least size 2^λ. Since the ϵ_c is chosen by the adversary before r is sampled, the adversary will be able to make the protocol accept an incorrect tuple with probability at most 2^λ. This argument corresponds to the proof of Claim 6 by Cramer et al. [CDE+18].

Efficient Batch Verification: The check can be batched for an arbitrary number of triples, which makes the cost for generating the random number r negligible (similarly to many other protocols in the literature that sacrifice a tuple to check another one).

Furthermore, one can reduce the communication for the final step of the protocol: If the parties hold $\{[\![x^{(j)}]\!]_m\}$ and would like to check if $x^{(j)} = 0$ for all j, every party P_i computes $x_i^{(j)} = 0 - x_{i-1}^{(j)} - x_{i+1}^{(j)}$. All parties then hash $\{x_0^{(j)}, x_1^{(j)}, x_2^{(j)}\}_{\forall j}$ and broadcast their result. If there is a mismatch, they abort. With these two optimizations, the asymptotic communication per triple is determined by the two optimistic multiplications and the opening of e. All involve sending one $m + \lambda$-bit value to one other party, so we arrive at $3(m + \lambda)$ bits per party and multiplication. If the protocol is used for preprocessing there are another m bits sent per party and multiplication during the online phase, otherwise $3(m + \lambda)$ is the total cost per multiplication. For the common choice of $m = 64$ and $\lambda = 40$, this gives a total of 312 bits.

7 Protocol with Postprocessing

Our postprocessing protocol is similar in spirit to our [CDE+18]-style preprocessing protocol from Section 6.2 in the sense that we use similar building blocks, but in a different order, which allows us to reduce the total

Protocol with Postprocessing

Input Each party P_i has input x_i.

---

1. Each party P_i secret shares its input x_i using the secret sharing with redundant shares as described in Section 3.4 to obtain $[\![x_i]\!]_{m,\lambda}$.
2. All parties jointly evaluate the desired circuit, where multiplications are performed optimistically as described in Section 3.3. Let $\{([\![x_i]\!]_{m,\lambda}, [\![y_i]\!]_{m,\lambda}, [\![z_i]\!]_{m,\lambda})\}_{i\in[N]}$ be the set of all the performed multiplications, where x_i, y_i are the left and right inputs and z_i is the output of the i-th multiplication gate.
3. For $i \in [N]$, compute a random $[\![a_i]\!]_{m,\lambda}$ and optimistically compute $[\![c_i]\!]_{m,\lambda} = [\![y_i \cdot a_i]\!]_{m,\lambda}$.
4. All parties jointly generate a random value $r \in \mathbb{Z}_{2^\lambda}$.
5. The parties reveal $[\![e_i]\!]_{m,\lambda} = r[\![x_i]\!]_{m,\lambda} + [\![a_i]\!]_{m,\lambda}$
6. For each $i \in [N]$ the parties check
$$r[\![z_i]\!]_{m,\lambda} + [\![c_i]\!]_{m,\lambda} - e_i[\![y_i]\!]_{m,\lambda} \stackrel{?}{=} 0$$
and abort if any of these checks fails.
7. If all checks passed, then the parties jointly open the output wires of the computed circuit.

Fig. 9: Protocol for secure circuit evaluation that does not require a preprocessing phase.

computation time at the cost of a slightly more expensive online phase. We describe our protocol in Figure 9 for the three party setting. For the sake of simplicity we describe the protocol with separate checks for each multiplication gate, but optimizations like the batch verification described in Section 6.2 can be applied equally well to this protocol. The security proof for the protocol is completely analogous to the security proof in Section 6.2.

8 Implementation and Evaluation

To help adoption and accessibility of our protocols, we implemented them using Sharemind [Bog13] and the MP-SPDZ framework [Dat]. While some protocols mentioned in this work are only implemented in either framework, we have implemented the basic semi-honest and the postprocessing protocol from Section 7 in both. We provide extensive benchmarks in both LAN and WAN settings for both implementations as well as a theoretical analysis of the asymptotic communication. Throughout this section, we use a statistical security parameter $\lambda = 40$.

8.1 Communication

Table 1 shows the communication complexity per multiplication in $\mathbb{Z}_{2^{64}}$ with the various protocols for statistical security parameter $\lambda = 40$. While the numbers are obtained from running the protocols in batches of at least one million with rounding, they match the asymptotic cost one would expect from a manual analysis. For comparison, we have added the figures reported in a recent concurrent work by Chaudhari et al. [CCPS19] (averaged over the parties because their protocol is asymmetric).

One optimistic multiplication in $\mathbb{Z}_{2^m}$ requires sending m bits, and using Beaver multiplication in the data-dependent phase requires opening two masked values, thus sending $2m$ bits. A CDE+18-style sacrifice [CDE+18] requires two optimistic multplications and one opening in $\mathbb{Z}_{2^{m+\lambda}}$, while ABF+17 [ABF+17] asymptotically requires three optimistic multiplications and two classic sacrifices that require two openings each.[10] This comes down to $7m$ bits. Finally, DOS18 preprocessing [DOS18] with SingleVerify requires two

[10] Because of cut-and-choose we cannot use the trick used for DOS18-style sacrificing.

	Preprocessing	Data-dep.	Total
DOS18 preprocessing (single)	992	128	1120
DOS18 preprocessing (batch)	464	128	592
ABF+17 preprocessing	448	128	576
CDE+18 preprocessing	312	128	440
Postprocessing	-	312	312
Semi-honest	-	64	64
Malicious ASTRA [CCPS19]	448	85	553

Table 1: Communication per party and $\mathbb{Z}_{2^{64}}$ multiplication (bits)

optimistic multiplications in $\mathbb{Z}_p$ and two openings in $\mathbb{Z}_p$ as well as sending two $m+\lambda$-bit values for sharing over the integers, totalling in $2(m+\lambda)+3\log p$ bits. It roughly holds that $\log p > 7+2m+3\lambda$, so for our choice of parameters $\log p > 255$.[11] The slight difference to the figure in the table comes from rounding up to multiples of eight. Using BatchVerify in Figure 7 allows to avoid the openings in $\mathbb{Z}_p$, bringing the complexity to slightly more than ABF+17 preprocessing.

8.2 Implementation in Sharemind

Sharemind already supported semi-honest computation in $\mathbb{Z}_{2^{32}}$ and $\mathbb{Z}_{2^{64}}$. We have added [DOS18]-style preprocessing with BatchVerify from Section 6.1 and postprocessing from 7.

8.3 Implementation in MP-SPDZ

MP-SPDZ already supported replicated secret sharing in $\mathbb{Z}_{2^{64}}$ and $\mathbb{Z}_p$ as well as the protocol for $\mathbb{Z}_p$ by Lindell and Nof [LN17]. Its use of C++ templating easily allows to add new protocols reusing existing components, and it provides an efficient implementation of $\mathbb{Z}_{2^k}$ arithmetic for any k. It also uses Montgomery representation for arithmetic modulo a prime.

We have implemented the following protocols: [DOS18]-style preprocessing with SingleVerify from Section 6.1, cut-and-choose preprocessing of triples similar to Araki et al. [ABF+17], and [CDE+18]-style preprocessing as well as postprocessing from Section 6.2.

8.4 Benchmarks

We have run our implementations in SIMD fashion, that is, combining the communication of a varying number of multiplications in as few network messages as possible. It is of little surprise that up to a certain number the throughput increases.

Figure 10 shows our benchmarks for various numbers of parallel multiplications in a LAN, that is AWS `c5.9xlarge` instances in the same region. This type features 36 virtual CPUs, 72 GiB of RAM, and 10 Gbit/s network network connection.

All benchmarks in this section are averages over ten executions. The figure for cut-and-choose is limited to 1048576 because the analysis by Araki et al. [ABF+17] mandates batches of at least this size. The plot shows that all malicious protocols perform similarly except the [DOS18] protocol, and that the postprocessing protocol is slightly ahead as one would expect.

Figure 11 shows our benchmarks for various numbers of parallel multplications in a continental WAN, that is one AWS `c5.9xlarge` instance in each of Frankfurt, London, and Paris. The results mirror the results in the LAN setting except for the fact that 2^{20} parallel multiplications perform better than 2^{15} for all protocols. This is most likely because of the increased network delay of up to 12 ms.

[11] According to Damgård et al. [DOS18], $p > 100 \cdot 2^{2m+2\lambda}$, but a quick recalculation of $24 \cdot B^2 2^\lambda$ with $B = 2^{m+\lambda+1}$ shows that it should be $3s$ instead of 2λ in the inequality for p.

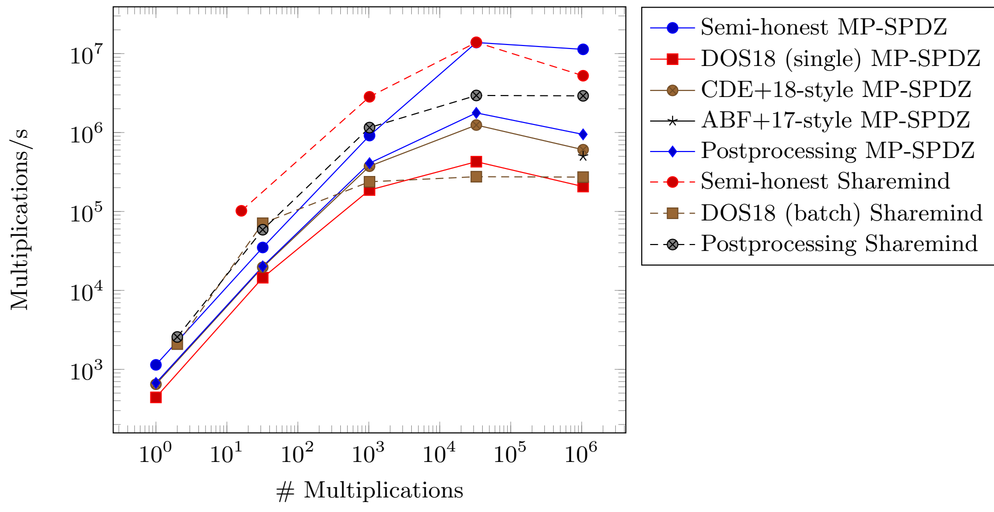


Fig. 10: Throughput in LAN

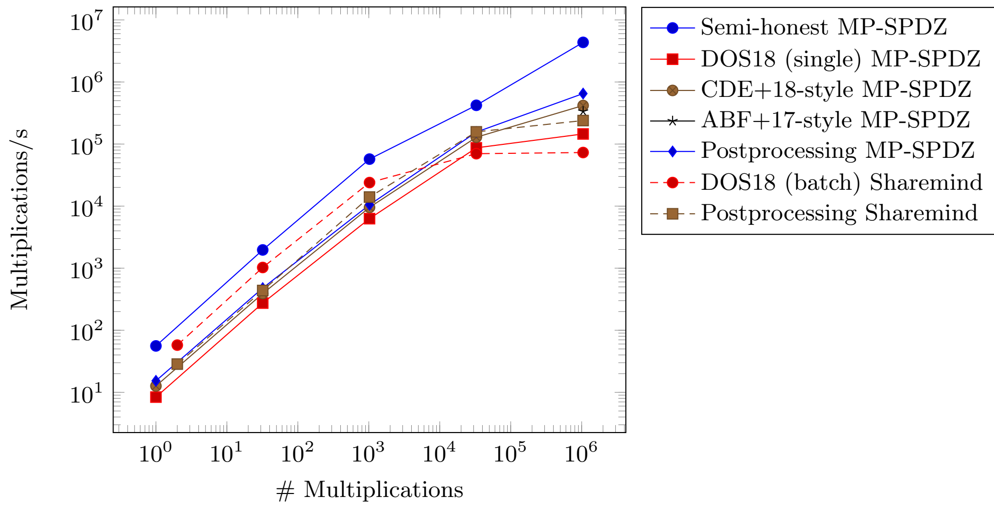


Fig. 11: Throughput in continental WAN

Finally, 12 shows our benchmarks for a global WAN, that is one AWS `c5.9xlarge` instance in each of Frankfurt, Northern California, and Tokyo. The largest network latency we observed is 236 ms in this setting.

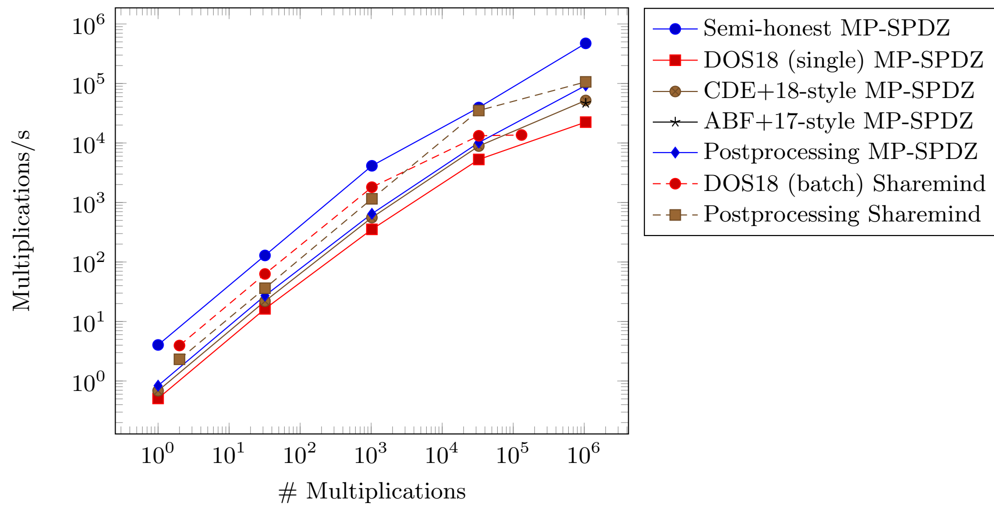


Fig. 12: Throughput in global WAN

8.5 Multithreading

We generally found that our protocols do not use all bandwidth that is available. Figure 13 supports this by showing that increasing from a single thread to 128 increases the output while keeping same the number of parallel multiplications.

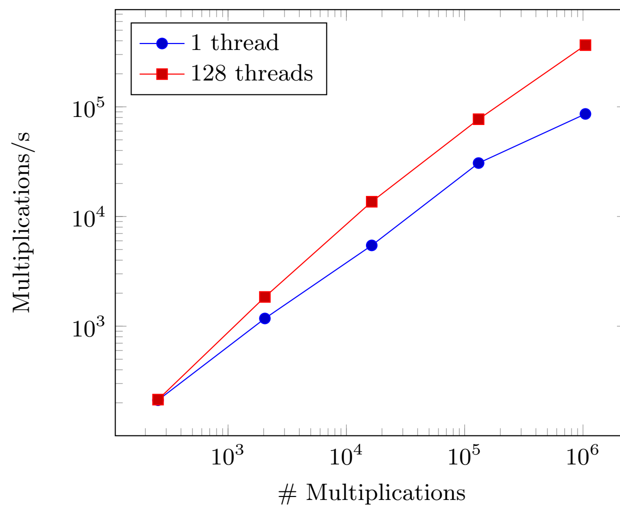


Fig. 13: Throughput with postprocessing on MP-SPDZ in global WAN

9 Comparison with Other Related Work.

In this section, we provide a comparison with the most relevant previous works.

The concurrent work of Choudhary et al. [CCPS19] does not provide throughput of multiplications for their offline phase, only for more complex computation such as AES evaluation. It is therefore hard to compare their implementation to ours. Furthermore, AES evaluation does not lend itself to computation in $\mathbb{Z}_{2^k}$ for $k > 1$, which makes a rather odd benchmark in this setting.

Three party honest majority actively secure multiplication with 61-bit Mersenne field is implemented in [CGH+18] where they measure that a circuit with 10^6 multiplication gates and depth 20 can be evaluated in 0.3 seconds in a single AWS region (presumably 10 Gbit/s networks). This amounts to a throughput of 3.3 million multiplications per second, while our postprocessing protocol with Sharemind in a LAN achieved 2.9 million for a slightly smaller batch size that $10^6/20$. This shows that our protocol is competitive despite the extra effort needed for rings as compared to fields.

The batchwise multiplication verification is optimized in [NV18]. The authors estimate that their computation optimizations achieves up to 10^7 two-party verifications per second using multithreading for 64-bit primes and up to $5 \cdot 10^6$ with 128-bit prime. Their estimations are based on their implementation of the computations and estimates for the communication but they do not benchmark the protocol with communication. From a conceptual point of view, [NV18] uses similar verification ideas as we do, hence their work indicates that our implementation might also benefit from more optimized field arithmetic. However, we still need to use larger fields to accommodate the integer secret sharing meaning we need more communication to achieve triples modulo 2^k of the same length as their modulo prime triples.

References

ABF+17. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

ABL+18. David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

ACD+19. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. Cryptology ePrint Archive, Report 2019/872, 2019. https://eprint.iacr.org/2019/872.

AKTZ17. Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, Vancouver, BC, 2017. USENIX Association.

BCD+09. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer, Heidelberg, Germany.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

BFO12. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

BJSV15. Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015: 19th International Conference on Financial Cryptography and Data Security*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234, San Juan, Puerto Rico, January 26–30, 2015. Springer, Heidelberg, Germany.

BKK+16. Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *PoPETs*, 2016(3):117–135, 2016.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

BNTW12. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Secur.*, 11(6):403–418, November 2012.

Bog13. Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

BTW12. Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *FC 2012: 16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64, Kralendijk, Bonaire, February 27 – March 2, 2012. Springer, Heidelberg, Germany.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

CCPS19. Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. Cryptology ePrint Archive, Report 2019/429, 2019. https://eprint.iacr.org/2019/429.

CDE+18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. Lecture Notes in Computer Science, pages 769–798, Santa Barbara, CA, USA, 2018. Springer, Heidelberg, Germany.

CDN15. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

CGH+18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 34–64, Cham, 2018. Springer International Publishing.

CRFG19. Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli. Monza: Fast maliciously secure two party computation on z_{2^k}. Cryptology ePrint Archive, Report 2019/211, 2019. https://eprint.iacr.org/2019/211.

Dat. Data61. MP-SPDZ - Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ.

DEF+19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning, 2019. https://eprint.iacr.org/2019/599.

DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

DO10. Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223

of *Lecture Notes in Computer Science*, pages 558–576, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

DOS18. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. Lecture Notes in Computer Science, pages 799–829, Santa Barbara, CA, USA, 2018. Springer, Heidelberg, Germany.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

FGMv02. Matthias Fitzi, Nicolas Gisin, Ueli M. Maurer, and Oliver von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 482–501, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, May 8–12, 2017. Springer, Heidelberg, Germany.

GIP+14. Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 495–504, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

JNO14. Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 81–92, 2014.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842, 2016.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 259–276. ACM Press, 2017.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS 18: 25th Conference on Computer and Communications Security*, pages 35–52. ACM Press, 2018.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing.

PL15. Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 75–89, 2015.

RB89. Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st Annual ACM Symposium on Theory of Computing*, pages 73–85, Seattle, WA, USA, May 15–17, 1989. ACM Press.

Sch18. Berry Schoenmakers. MPyC – Secure multiparty computation in Python. GitHub, 2018. https://github.com/lschoe/mpyc.