Use your Brain! Arithmetic 3PC For Any Modulus with Active Security[⋆]

Hendrik Eerikson[1], 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)

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 three-party computation with an honest majority, which is among the most popular models for real-world applications of MPC. Somewhat surprisingly, despite its significant popularity, there are currently no practical solutions for evaluating 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. Existing solutions are either only passively secure or require the computations to be performed over prime fields, which do not match real-world system architectures. This is unfortunate, since it requires application developers to redesign their applications for the models of computation that are provided by existing MPC frameworks, rather than the MPC frameworks matching the needs of the developers.
In this paper we present the first fully-fledged implementation of an MPC framework that can evaluate arithmetic circuits with arbitrary word sizes. Our framework is based on a new protocol, which improves the communication overhead of the best known previous solutions by a factor of two. We provide extensive benchmarks of our framework in a LAN and in different WAN settings, showing that the online overhead for achieving active security is less than two, when compared to the best solutions for the same setting with passive security. Concretely, for the case of 32- and 64-bit words, we show that our framework can evaluate 10^6 multiplication gates per second.

"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

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 $x_1, \ldots, x_n$ 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

⋆ 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.

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 with unconditional security, i.e. protocols whose security does not depend on any unproven computational assumptions and whose security holds regardless of the computing power of the adversary. Moreover, protocols with information-theoretic security are very often more efficient than their computationally secure counterparts, since the latter class of protocols necessarily relies on expensive "public key" operations like exponentiations, while the former class of protocols typically only uses light-weight arithmetic operations.

Existing implementations of three-party computation protocols with information-theoretic security 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 present a novel compiler that is more efficient than the one of Damgård et al. [DOS18]. 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[4] 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 messages are sent redundantly, thus incurring a multiplicative blow-up in the bandwidth overhead of the protocol.

From a conceptual point of view, our main contribution is a new method that significantly reduces the number of redundant messages that need to be sent during a protocol execution. The idea behind our

[4] 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.

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 3. 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.

In Section 4, we present the concrete instantiation of our compiler that we implemented in our MPC framework. In Section 6, we provide extensive performance benchmarks of our framework, both in the LAN as well as different WAN settings. Our framework constitutes the first implementation of an three-party computation protocol for arithmetic circuits modulo 2^{32} and 2^{64} with active security. Our framework is built on top of the Sharemind MPC platform and we also provide benchmarks comparing their passively secure protocol suite to our actively secure one. Like many previous works, our protocols is split into a preprocessing phase, during which we generate some form of correlated randomness, and an online phase, during which the preprocessed correlated randomness is used to perform the actual computation. Our preprocessing protocol is described in Section 5 and in Section 6 we highlight some of the practical optimizations like batch processing techniques and discrete Fourier transform that helped improving the performance of our preprocessing.

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 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-theoretic *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 Damgård et al. [CDE+18], this limitation has been lifted, allowing to perform computation modulo 2^k (by defining the MACs modulo 2^{k+s} where s is the security parameter, thus introducing an overhead proportional to the security parameter). It is currently not known how to extend their approach to the honest-majority case. An implementation (and optimizations) of [CDE+18] was presented in [DEF+19].

Other recent works have considered active security in the three-party setting: For Boolean circuits, [FLNW17] uses correlated random number generation to achieve efficient preprocessing and replication to achieve active security. 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 verity correctness (their approach can also be used for more than three parties).

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. The check returns 1 if the values are equal and 0 if not. 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

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] (essentially the property was defined also 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.

2.2 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.

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

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 $\Theta(\lambda)$.

2.3 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 4 and the modulus m defines the word size over which computations will be performed. For example, for arithmetic computations over 64-bit integers, one can set $m = 2^{64}$. 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 $a \in \mathbb{Z}_m$, it picks uniformly random $a_1, a_2 \leftarrow \mathbb{Z}_m$, sets $a_3 = a - a_1 - a_2 \mod m$, and sends a_j to each P_j. We write $[a]_m$ to denote an additive secret sharing of a modulo m.

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}.

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 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

Addition by a constant: To add constant c to $[a]_m$, i.e., compute $[b]_m$ with $b = c + a \mod m$, we let P_1 locally compute $b_1 = a_1 + c \mod m$, 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 $c = a + b \mod m$, every party P_i locally adds its shares, i.e., every party computes $c_i = a_i + b_i \mod m$.

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

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

2.4 Additive Replicated Secret Sharing

We will use additive replicated secret sharing in our preprocessing protocol in Section 5. 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 $a \in \mathbb{Z}_m$, it picks $a_1, a_2 \leftarrow \mathbb{Z}_m$, sets $a_3 = a - a_1 - a_2 \mod m$, and sends a_{j-1} and a_{j+1} to each P_j. Parties P_{i-1} and P_{i+1} send each other the value a_i they received, verify that their values are consistent, and abort if not. We write $[\![a]\!]_m$ to denote an additive replicated secret sharing of a modulo m.

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 $a = a_1 + a_2 + a_3 \mod m$ if the check passed.

Addition by a constant: To add a public constant c to a secret shared value $[\![a]\!]_m$, i.e., to compute $[\![b]\!]_m$, where $b = c + a \mod m$, 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 $c = a + b \mod m$ every party P_i locally adds their shares, i.e., it computes $c_{i-1} = a_{i-1} + b_{i-1} \mod m$ and $c_{i+1} = a_{i+1} + b_{i+1} \mod m$.

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

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

1. Each P_i picks $s_i \leftarrow \mathbb{Z}_p$ and computes $u_i = a_{i+1}b_{i+1} + a_{i+1}b_{i-1} + a_{i-1}b_{i+1} + s_i$
2. Send u_i to P_{i+1} and s_i to party P_{i-1};
3. P_i receives u_{i-1} as well as s_{i+1} and defines its shares of $[\![c]\!]_m$ as $c_{i-1} = u_i - s_{i+1}$ and $c_{i+1} = u_{i-1} - s_i$

2.5 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 $a \in \mathbb{Z}_m$ over the integers using shares with bit-length $\log m + \lambda$. The λ extra bits ensure that the statistical distance between the distributions of shares for any two values in $\mathbb{Z}_m$ is negligible in λ.

Sharing a value: To share a value $a \in \mathbb{Z}_m$, P_i picks $a_1, a_2 \leftarrow \{0, \ldots, 2^{\lceil \log m \rceil + \lambda} - 1\}$ 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. Each P_i picks $s_i \leftarrow \{0, \ldots, 2^{2\lceil \log B \rceil + \lambda + 2} - 1\}$ and computes $u_i = a_{i+1}b_{i+1} + a_{i+1}b_{i-1} + a_{i-1}b_{i+1} + s_i$.
2. Send u_i to P_{i+1} and s_i to party P_{i-1}.
3. P_i receives u_{i-1} as well as s_{i+1} and checks that $|u_{i-1}| \leq 2^{2\lceil \log B \rceil + \lambda + 3}$ and $|s_{i+1}| \leq 2^{2\lceil \log B \rceil + \lambda + 2}$.
4. P_i defines its shares of $[\![c]\!]_p$ as $c_{i-1} = u_i - s_{i+1}$ and $c_{i+1} = u_{i-1} - s_i$.

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

3 Extension of the Compiler by Damgård et al.

The compiler $\mathsf{COMP}_{\mathsf{old}}$ by Damgård et al. [DOS18] takes an n-party passively $\left(t^2 + t\right)$-secure protocol $\mathit{\Pi}$ and transforms it into a protocol $\mathsf{COMP}_{\mathsf{old}}\left(\mathit{\Pi}\right)$ that is secure with abort against t active corruptions[5]. 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 execute the passively secure protocol on behalf of the real parties. Each virtual party $\mathbb{P}_i$ is simulated by $t+1$ real parties $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$ 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 $\mathit{\Pi}$. 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 $\mathit{\Pi}$. In the second phase, the virtual parties run $\mathit{\Pi}$ 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 $\mathit{\Pi}$, *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 $\mathit{\Pi}$ sends ℓ messages during a protocol execution, then $\mathsf{COMP}_{\mathsf{old}}\left(\mathit{\Pi}\right)$ will send roughly $\mathcal{O}\left(\ell \cdot t^2\right)$ many messages.

[5] 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.

3.1 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 $\mathit{\Pi}$, 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 $\mathit{\Pi}$ and outputs a compiled protocol $\mathsf{COMP}_{\mathsf{new}}\left(\mathit{\Pi}\right)$ that is secure against t active corruptions. If $\mathit{\Pi}$ sends ℓ messages in total, then our compiled protocol will only send $\mathcal{O}\left(\ell \cdot t + t^2\right)$ messages.

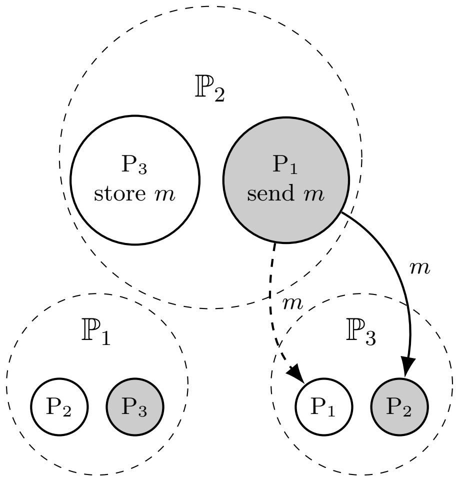


Fig. 3: An illustration of our simulation strategy for the case of three parties with one active corruption. Dashed circles represent virtual parties. Solid circles inside the dashed circles represent the real parties that simulate the given virtual party. The brains of each virtual party are highlighted 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 $\mathit{\Pi}$. 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 $\mathit{\Pi}$, 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 $\mathit{\Pi}$ 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 $\mathit{\Pi}$. During the computation phase of $\mathit{\Pi}$, 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

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

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 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.

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 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 $f\left(x_1, \ldots, x_n\right)$ 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 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 $\mathsf{different}$, 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.

4 Efficient Three-Party Computation

Using our new compiler, we construct the currently most efficient three-party protocol for evaluating arithmetic circuits over arbitrary rings that is secure against one active corruption. Towards this goal, we apply our compiler to the passively secure circuit evaluation approach of Beaver [Bea92]. The correlated randomness required by Beaver's protocol is generated in a separate preprocessing phase, which is described in Section 5.

4.1 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 $\mathbb{Z}_m$ 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 $c_i = a_i \cdot b_i \mod m$. 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 2.3. 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.*

4.2 Our Protocol

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 $x_i \in \mathbb{Z}_m$. 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[6]. Our concrete preprocessing protocol will be described in detail in Section 5. Since for $1 \leq i \leq 3$,

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

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

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

1. **Input sharing:**
   (a) Each P_i picks $x_i^1, x_i^2 \leftarrow \mathbb{Z}_m$ and sets $x_i^3 = x_i - x_i^1 - x_i^2 \mod m$.
   (b) For $1 \le j \le 3$, 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 $\mathbb{Z}_m$ with active security with abort against one active corruption.

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 $\mathsf{COMP}_{\mathsf{new}}\left(\mathsf{Beaver}_{f'}\right)$ 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 $\log(m)$ 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 $1280 + 4\log(m)$ bits.

5 Preprocessing

During the preprocessing phase we generate replicated secret sharings of multiplication triples $c = a \cdot b \mod m$. Our preprocessing protocol combines the processing 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 m to obtains shares of multiplication triples in our desired ring $\mathbb{Z}_m$[7].

[7] Valid multiplication triples over integers are valid multiplication triples modulo m.

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.

5.1 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 $a_i, b_i \in \mathbb{Z}_m$. 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 2.5 to compute $[\![c]\!]_{\mathbb{Z}}$.

5.2 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 holds in $\mathbb{Z}_p$, then it also holds over the integers, since p is chosen so large that no wrap around modulo p happens during the multiplication.

5.3 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 $i \in \{1, \dots N\}$, 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 *single* invocation of SingleVerify. The main idea behind their approach is to encode all multiplication triples $(a_1, b_1, c_1), \dots, (a_N, b_N, c_N)$ 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 $a_1, \dots, a_N$ and $b_1, \dots, b_N$. 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 $i \in \{1, \dots N\}$, we set $h(i) = c_i$. For $i \in \{N+1, \dots 2N-1\}$, 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 $i \in \{1, \dots, N\}$ of polynomial f and we would like to evaluate f on point z. Define $\delta_i^N(x)$ 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 m. The concrete efficiency of this protocol is discussed in Section 6.

6 Implementation and Evaluation

We implement our protocol with three parties and one active corruption on top of the Sharemind multi-party computation platform [Bog13]. The main protocol suite for Sharemind is called `shared3p` [BNTW12] and considers three party computation with one passive corruption. Their protocol is based on additive secret sharing modulo 2^k. In the following we refer to our protocol as `shared3a`.

Implementing the Protocol. In our implementation, randomness is generated using pseudorandom number generators. This allows us to just send short seeds instead of full random tapes during the first phase of the protocol.

Our implementation is in the client-server model for MPC, meaning that parties other than the computing parties can share their inputs through a client application. Upon receiving the inputs, the compute parties compare replicated shares within virtual parties as dictated by the ideal functionality $\mathcal{F}_{\mathsf{bcast}}$. If a party finds a discrepancy, it stops sending and receiving network messages. In such a case, it is up to the remaining honest party to notice and also stop its execution of the protocol.

To implement $\mathcal{F}_{\mathsf{check}}$, every real party P_i keeps track of five hashes of messages. For virtual party $\mathbb{P}_{i+1}$, where P_i is the brain, two hashes of messages are kept. Hash h_1 is of messages received from $\mathbb{P}_{i-1}$ and h_2 is of messages received from $\mathbb{P}_i$. For the virtual party $\mathbb{P}_{i-1}$, where P_i is the pinky, three hashes of messages are kept. Hashes h_3 and h_4 consist of messages sent to and received from $\mathbb{P}_i$ respectively. Hash h_5 is of messages sent to $\mathbb{P}_{i+1}$. Messages received from $\mathbb{P}_{i+1}$ are not transcribed since the sending real party is P_i.

To verify that each party followed the protocol description honestly, a real party P_i has to perform the following comparisons: compare h_1 with P_{i+2}'s h_4, h_2 with P_{i+1}'s h_3, h_3 with P_{i+2}'s h_2, h_4 with P_{i+1}'s h_1 and h_5, h_5 with P_{i+2}'s h_4. Additionally P_i has to compare its hashes h_1 and h_5. If any of the pairs of hashes are not equal, the party aborts and stops responding to network messages.

We implemented the preprocessing using the Lagrange interpolation idea directly as in Figure 7 and also using fast Fourier transform as in Figure 8 as inspired by [NV18]. We assume that ω is a 2N'th primitive root of unity and hence ω^2 is an N'th root of unity in $\mathbb{Z}_p$. Hence, we consider only the cases where N is

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 an 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.

a power of two. Compared to the BatchVerify our FFTBatchVerify changes the local computations and the locations of the polynomials where the input triples are encoded.

We implemented the protocols for 32 and 64-bit unsigned and signed integers. Note that unsigned k-bit integers directly give us signed $k-1$-bit integers in two's complement notation where the most significant bit defines the sign. Moreover, addition and multiplication protocols remain exactly the same as for unsigned integers. Hence, we only consider benchmarks of the unsigned integers.

For the triple generation and verification we need to choose a statistical security parameter λ. Following the standard in the field, we mainly use $\lambda = 40$ (meaning that the protocol is secure except with probability 2^{-40}, regardless on the computing power of the adversary). To study the effect of the security parameter on the efficiency of the protocol, we also perform some benchmarks with $\lambda = 80$.

According to the bounds given in [DOS18], when $\lambda = 40$ we use a 214-bit prime, while for $\lambda = 80$ we use a 294-bit prime. We use The GNU Multiple Precision Arithmetic Library (GMP) for field arithmetic in the preprocessing phase. In addition, we use SHA256 hash to verify the transcripts.

FFTBatchVerify

Inputs: Parties P_1, P_2, and P_3 share N preprocessed triples $\llbracket a_i \rrbracket_{\mathbb{Z}}, \llbracket b_i \rrbracket_{\mathbb{Z}}, \llbracket c_i \rrbracket_{\mathbb{Z}}$ for $i \in \{1, \ldots, N\}$ over the integers.
Let $\llbracket x \rrbracket_p, \llbracket y \rrbracket_p, \llbracket z \rrbracket_p$ be an 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 $\llbracket f(\omega^{2(i-1)}) \rrbracket_p := \llbracket a_i \rrbracket_p$ and $\llbracket g(\omega^{2(i-1)}) \rrbracket_p := \llbracket b_i \rrbracket_p$.
2. Locally interpolate $f(x)$ and $g(x)$ (on N'th roots of unity) to get the coefficients $\llbracket f_i \rrbracket_p$ and $\llbracket g_i \rrbracket_p$ for $i \in \{1, \ldots, N\}$.
   For $i \in \{0, \ldots, 2N-1\}$, compute $\llbracket f(\omega^i) \rrbracket_p$ and $\llbracket g(\omega^i) \rrbracket_p$ using FFT on $2N$'th roots of unity
3. For $i \in \{1, \ldots, N\}$, define $\llbracket h(\omega^{2(i-1)}) \rrbracket_p := \llbracket c_i \rrbracket_p$.
4. For $i \in \{1, \ldots, N\}$, compute $\llbracket h(\omega^{2i}) \rrbracket_p = \llbracket f(\omega^{2i}) \rrbracket_p \cdot \llbracket g(\omega^{2i}) \rrbracket_p$ optimistically.
   Locally interpolate $h(x)$ using FFT on $2N$'th roots of unity to get the coefficients $\llbracket h_i \rrbracket_p$ for $i \in \{0, \ldots, 2N-1\}$.
5. Every party P_i picks a random $\llbracket z_i \rrbracket_p$.
6. The parties compute $\llbracket z \rrbracket_p = \sum_{i=1}^{3} \llbracket z_i \rrbracket_p$ and open z.
7. Compute locally using Horner's rule and the coefficients of the polynomials
$$\begin{aligned}\llbracket \alpha \rrbracket_p &= \llbracket f(z) \rrbracket_p, \text{ and}\\ \llbracket \beta \rrbracket_p &= \llbracket g(z) \rrbracket_p, \text{ and}\\ \llbracket \gamma \rrbracket_p &= \llbracket h(z) \rrbracket_p\end{aligned}$$
8. Check SingleVerify $(\llbracket \alpha \rrbracket_p, \llbracket \beta \rrbracket_p, \llbracket \gamma \rrbracket_p, \llbracket x \rrbracket_p, \llbracket y \rrbracket_p, \llbracket z \rrbracket_p)$.

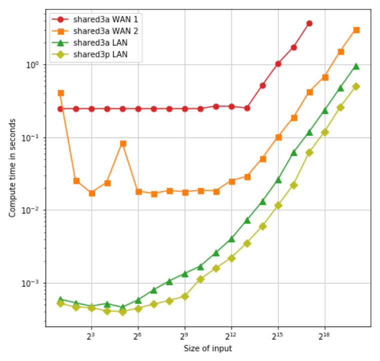
Fig. 8: Batch verification using FFT

6.1 Benchmarks of the implementation

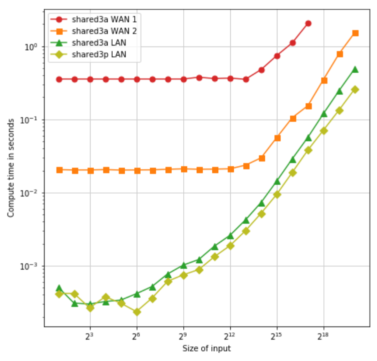
The benchmarks were performed in three settings: a local area network (LAN) and two wide area networks (WANs). The LAN setting consists of a cluster of three machines, where each machine has a dedicated 1 Gbit/s network link, 48 GiB of RAM and two Intel Xeon X5670 2.93 GHz 6 core / 12 thread processors. In total each machine has 24 parallel threads with Intel HyperThreading. The network latency between two cluster nodes in the LAN is 0.18 ms. The two WANs consist of three Amazon Web Services EC2 c5.9xlarge instances. Each instance has 36 virtual CPUs, 72 Gib of RAM and a 10 Gbit/s network connection. In WAN 1 we chose instances, which span across different continents and concretely our instances are in Frankfurt, Northern California, and Tokyo. The largest network latency we observed is 236 ms. In WAN 2 we chose instances on the same continent, namely in Frankfurt, London, and Paris (as can be observed in Figure 9a and 9d, experiments performed in this setting are particularly noisy, probably due to AWS shared hardware). The largest observed latency is 12 ms.

The Sharemind protocol as well as our protocol, operate on vectors in a SIMD fashion. For example, multiplication of two vectors means pairwise multiplication of the elements in the same coordinates. Hence we measure the efficiency for various lengths of the input vector to show that we benefit from such parallelization because sending separate messages for each operation includes more network overhead. The benchmarking results we present here are averages over 10 runs and exact times can be found in Appendix A.

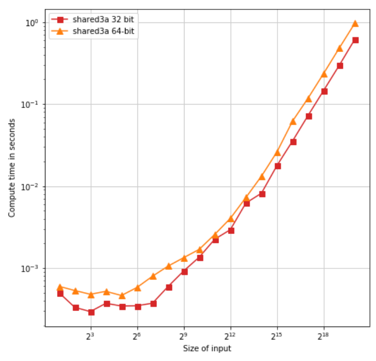
Comparison with Sharemind's Passively Secure Protocol. We compare the benchmarks of `shared3a` to those of the development version of Sharmind's passively secure protocol `shared3p`. Fig. 9a summarizes our mul-



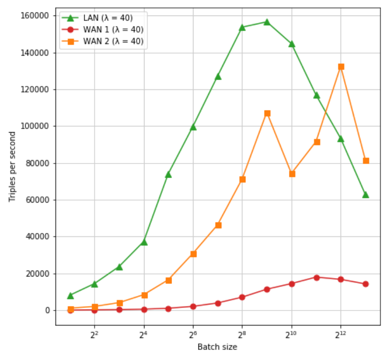
(a) Online Multiplications, 64-bit: `shared3a` vs. `shared3p`.



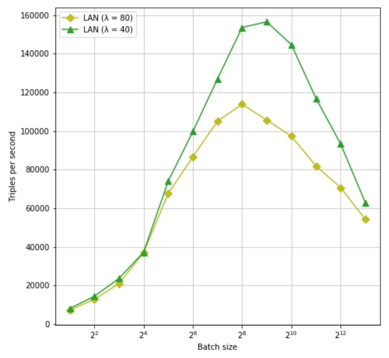
(b) Opening Phase, 64-bit values: `shared3a` vs. `shared3p`.



(c) Online Multiplication, LAN, 32- vs. 64-bit.



(d) Preprocessing Multiplications via FFT,
64-bit values: batch sizes and network.



(e) Preprocessing Multiplications via FFT,
64-bit values, LAN: 40- vs. 80-bit security.

Fig. 9: Experimental Results

tiplication benchmarks for 64-bit integers. On average the online phase of multiplication of 64-bit values is 1.94 times slower on `shared3a` than on `shared3p` which is expected as our protocol sends twice as many messages of the same length as required by the passive protocol. Compared to LAN, the multiplication in WAN setting is significantly less efficient for smaller input sizes that are more affected by latency but we can still achieve more than 32000 multiplications per second using parallelization. Fig. 9c shows the efficiency of the online multiplication as a function of the word size (32- vs. 64-bit).

The opening phase with 64-bit values is 1.79 times slower on `shared3a` than on `shared3p` as can be seen in Fig. 9b. In the passively secure protocol the shares of the output are simply sent between the parties. The benchmark of our protocol contains the check of the transcript as well as checking the received shares of the output. The comparison shows that these additional checks incur a very minor overhead in practice. Similarly to multiplication we can see that smaller input sizes are affected by latency and it is more efficient to use the SIMD style parallelization.

Fig. 9d and Fig. 9e summarize the efficiency of our preprocessing phase for 64-bit triples where local computation uses fast Fourier transform. The batch size denotes how many triples are verified together at once. Fig. 9e shows how the efficiency of the protocol is affected by the statistical security parameter λ in the same network condition, while Fig. 9d shows the dependency on the network condition (for a fixed security parameter $\lambda = 40$). The local computation time of our preprocessing protocol dominates our total timings for large batch sizes. In the LAN setting, we get the most efficient results for a batch size of 2^9, whereas the best batch sizes are slightly higher in WAN settings. The `shared3p` protocol does not require a preprocessing phase, but there are many other protocols that do (see below). Combining online and offline phase shows us that `shared3a` is approximately 15 times slower than `shared3p`.

Comparison with Other Related Work. For the sake of completeness, we provide a rough comparison of our implementation with other existing MPC implementations. We note, however, that here we are comparing "apples to oranges", since the protocols differ in the security they provide, the model of computation, and the number of parties they support. The aim of this comparison is to provide a rough idea of how our protocol compares to protocols in related settings.

The two-party SPDZ-based protocol of Keller et al. [KSS13] performs up to $6 \cdot 10^5$ multiplications in a 128-bit prime field per second in a LAN setting. Our three-party protocol implementation can perform more than 10^6 multiplications per second in a 64-bit ring. Both `shared3a` and SPDZ use the same multiplication protocol with preprocessed triples, but the number of parties and the structure of the shares varies resulting in different network messages.

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). Previous work [LN17] achieved the same computation with 0.5 seconds using a preprocessing approach. The depth of the circuit indicates that not all multiplications are calculated in parallel making it difficult to compare to our benchmarks. However, knowing that we achieve the same number of multiplications per second using a slower network it seems likely that our implementation can outperfom this result if it manages to take full advantage of the network capacity.

We also compare the efficiency of our preprocessing phase with those of other actively secure MPC protocols. MASCOT [KOS16] does preprocessing with OT and achieves about $2 \cdot 10^3$ triples per second for honest minority case for three parties with 1 Gbit/s network and 128-bit field. Overdrive [KPR18] does preprocessing with additively homomorphic encryption and proofs of knowledge. For two parties and 1 Gbit/s network, they achieve $2.3 \cdot 10^3$ to $5.9 \cdot 10^4$ triples per second depending on the choice of the security parameter and field size. For example, their best results are $3 \cdot 10^4$ for 64-bit security and 1 500 for 128-bit security and 128-bit field. They also showcase that the throughput decreases as the number of parties increases but there are no concrete results for three parties. Our implementation compares favourably with these.

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.

Communication Overhead of Preprocessing. For a prime modulus p of length $\log p$ the verification in $\mathsf{BatchVerify}$ requires $2N \log p + 20 \log p$ bits of communication where $4 \log p$ come from the opening of the random value z and $16 \log p$ from the publishing in the `SingleVerify`. The triple generation with optimistic multiplication takes $10 \log p$ per triple as four messages are needed to generate the random values and two to do optimistic multiplication. We need $N + 1$ triples for verification. The extra triple is generated in the field $\mathbb{Z}_p$ and takes $14 \log p$ bits. Hence, this gives us about $12 \log p + \frac{34}{N} \log p$ bit per triple which is around 8kbit of communication per triple for each party in our implementation for 64-bit triples. In comparison, SPD$\mathbb{Z}_{2^k}$ [CDE+18] has a bit over 300kbit of communication to achieve 64-bit triples with 64-bit statistical security. Overdrive [KPR18] reports other SPDZ protocol variants as requiring 9kbit to 350kbit for 64-bit secrets.

References

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.

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.

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.

DEF$^+$19. Ivan Damgård, Daniel Escudero, Tore Frederiksen, Peter Scholl, Nikolaj Volgushev, and Marcel Keller. New primitives for actively-secure mpc mod 2^k with applications to private machine learning. Cryptology ePrint Archive, Report 2019, 2019. https://eprint.iacr.org/2019/.

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.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 158–189, 2018.

KSS13. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 549–560, New York, NY, USA, 2013. ACM.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 259–276, New York, NY, USA, 2017. ACM.

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`.

A Additional Benchmarking results

Table 1 gives the exact results of the preprocessing times as triples per second. We can see that the straightforward implementation using Lagrange interpolation is the local computation is significantly less efficient than using discrete fast Fourier transform.

Table 2 gives the timings for the online part of the multiplication and Table 3 gives the times for the combination of the opening and transcript verification.

The empty cells in the table mean that we did not perform these benchmarks due to time constraints.

Table 1: Comparison of speed of precomputation (triples per second) with different batch sizes using Lagrange Interpolation or discrete fast Fourier transform (DFFT).

Batch size	Lagrange Interpolation LAN ($\lambda = 80$)	DFFT LAN ($\lambda = 80$)	DFFT LAN ($\lambda = 40$)	DFFT WAN 1 ($\lambda = 40$)	DFFT WAN 2 ($\lambda = 40$)
2	8045.08	7198.18	8051.13	0.00	1031.65
4	11897.01	12789.59	14353.64	127.41	1983.95
8	15752.94	20924.63	23610.90	248.86	4042.92
16	10500.36	37033.91	37207.81	503.82	8354.06
32	6539.40	67674.38	74045.28	1011.99	16440.33
64	3441.64	86613.59	99719.68	2014.44	30762.40
128		105130.54	127036.99	3875.03	46255.42
256		113845.80	153671.78	7030.88	71233.92
512		105670.66	156570.82	11349.28	107473.97
1024		97314.77	144744.72	14400.28	74233.13
2048		81824.27	116864.30	17912.72	91542.65
4096		70643.64	93288.24	16698.98	132432.94
8192		54255.89	62879.91	14243.32	81674.45

Table 2: Comparison of speed multiplication (in microseconds) with different input sizes.

Input size	`shared3p` LAN 64-bit	`shared3a` LAN 32-bit	`shared3a` LAN 64-bit	`shared3a` WAN 1 64-bit	`shared3a` WAN 2 64-bit
2^1	523	498	598	247544	410134
2^2	467	332	532	247358	25891
2^3	456	294	479	247442	17322
2^4	413	371	520	247460	23960
2^5	402	344	463	247237	84316
2^6	449	347	580	247149	18224
2^7	508	373	796	247177	16996
2^8	571	596	1057	247216	18518
2^9	651	922	1339	247424	17876
2^{10}	1119	1358	1692	247600	18743
2^{11}	1583	2252	2578	268193	18429
2^{12}	2217	2938	4034	265383	25397
2^{13}	3542	6212	7252	252819	28971
2^{14}	6014	8178	13189	519608	51636
2^{15}	11563	17825	26144	1027608	101967
2^{16}	22457	35464	62019	1732841	188045
2^{17}	61860	72716	118243	3670108	420561
2^{18}	119478	146689	236665		685853
2^{19}	260481	296340	481219		1529646
2^{20}	499197	615839	966447		3067342

Table 3: Comparison of speed declassify (in microseconds) with different input sizes.

Input size	`shared3p` LAN 64-bit	`shared3a` LAN 64-bit	`shared3a` WAN 1 64-bit	`shared3a` WAN 2 64-bit
2^1	417	504	355952	20560
2^2	416	307	355872	20312
2^3	264	298	355845	20336
2^4	375	322	355790	20574
2^5	306	337	355808	20255
2^6	235	412	355879	20328
2^7	359	517	355804	20419
2^8	609	768	355863	20769
2^9	747	1014	355931	21056
2^{10}	882	1211	376702	20784
2^{11}	1322	1845	361400	20875
2^{12}	1874	2591	366684	21071
2^{13}	3003	4218	353280	23624
2^{14}	5201	7305	479142	29743
2^{15}	9427	14289	743938	56271
2^{16}	18844	28873	1113496	105432
2^{17}	38044	56802	2056206	153704
2^{18}	71308	120031		350783
2^{19}	134282	247580		790954
2^{20}	259807	492332		1530294