An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings

Mark Abspoel[1,2], Anders Dalskov[3], Daniel Escudero[3], and Ariel Nof[4]

[1] Centrum Wiskunde & Informatica (CWI), Amsterdam, the Netherlands
[2] Philips Research, Eindhoven, the Netherlands
[3] Aarhus University, Denmark
[4] Bar Ilan University, Israel

Abstract. MPC over rings like $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ has received a lot of attention recently due to their potential benefits in implementation and performance. Several protocols for active security over these rings have been proposed recently, including an implementation in the dishonest majority setting due to Damgård et al. (S&P 2019) and in the popular three-party and one corruption setting. However, to this date, no concretely-efficient protocol for arithmetic computation over rings in the honest majority setting, which works for any number of parties, have been proposed.

In this work, we present a *new* compiler for MPC over the ring $\mathbb{Z}_{2^k}$ in the honest majority setting, that takes several building blocks, which can be essentially instantiated using semi-honest protocols, and turn them into a maliciously secure protocol. Our compiler follows the framework of Chida et al. (CRYPTO 18) for finite fields, and makes it compatible for rings using techniques from the work of Cramer et al. (CRYPTO 18), with only small additional overhead. Per multiplication gate, our compiler requires only two invocations of a semi-honest multiplication protocol over the larger ring $\mathbb{Z}_{2^{k+s}}$, where s is the statistical security parameter. As with previous works in this area aiming to achieve high efficiency, our protocol is secure with abort and does not achieve fairness, meaning that the adversary may receive output while the honest parties do not.

We provide two efficient instantiations for our compiler. The first instantiation is for the three-party case and is based on replicated secret sharing, where the resulting protocol requires each party to send just $2(k+s)$ bits per multiplication gate. To the best of our knowledge, this is the most efficient three-party protocol for large rings to this date. Our second instantiation is for any number of parties. In this case we manage to instantiate our compiler with a variant of Shamir secret sharing that was recently proposed by Abspoel et al. (TCC 2019). We show that the theoretical constructions from Abspoel et al. (TCC 2019) can be instantiated efficiently and prove that they satisfy the properties required by our building blocks. The resulting protocol requires each party to send just $14(k+s)\log n$ bits per multiplication gate. To the best of our knowledge, this is the *first* concretely-efficient protocol for MPC over rings with an honest majority that works for any number of parties.

We implemented our two protocols, run extensive experiments to measure their performance and report their efficiency. Our results prove that efficient honest-majority MPC over rings is possible.

1 Introduction

Multiparty Computation (MPC) is a cryptographic tool that allows multiple parties to compute a given function on private inputs whilst revealing only its output; in particular, parties' inputs and the intermediary values of the computation remain hidden. MPC has by now been studied for several decades, and different protocols have been developed throughout the years. These protocols can classified based on two orthogonal properties: the capabilities of the adversary, and how many parties the adversary can corrupt.

With regard to the former, the adversary is typically specified as either *actively* or *passively* corrupting parties. A passive adversary (also called *honest-but-curious* or *semi-honest* in the literature) tries to learn additional information by inspecting the protocol transcript, but is otherwise forced to follow the protocol as specified. An active adversary on the other hand is allowed to arbitrarily deviate from the protocol description. With regard to the latter, the two main settings usually considered are honest majority (where the adversary can control strictly less than half of the parties participating in the protocol) and dishonest majority (where the adversary can control any number of parties). In the dishonest majority setting, it has been shown that the security cannot be unconditional (i.e. it must rely on computational assumptions) and furthermore, an active adversary may cause the protocol to abort. On the other hand, when the adversary controls a minority of the participants, unconditional security is achievable and, furthermore, guaranteed output delivery (i.e. all parties get output in spite of arbitrary adversarial behavior) can be ensured.

Most MPC protocols are "general purpose", meaning that they can in principle compute *any* function. This generality is typically obtained by representing the function as an arithmetic circuit modulo some integer p. Note that implied in this representation, is a set of integers on which computation can be performed. Traditionally, MPC protocols are classified as being either *boolean* or *arithmetic*, where the former have $p = 2$ and the latter has $p > 2$. However, most of the existing arithmetic MPC protocols, independently of their security, require the modulus to be a prime (and for some protocols this prime must be large) [BTH06,BFO12,DKL+13,KOS16,FLNW17,CGH+18,LN17].

It was only until very recently that practical protocols in the arithmetic setting for a non-prime modulus were developed. The SPD$\mathbb{Z}_{2^k}$ protocol securely evaluates functions in the dishonest majority case [CDE+18], and several other works focus on honest majority case [FLNW17,CGH+18,ACD+19,EKO+19]. Computation over $\mathbb{Z}_{2^k}$ is appealing due to its practical benefits compared to computation over fields, as verified in [DEF+19], and it has been used in many applications of MPC [MR18,MZ17,BEDK19,WGC18]. This stems from the fact that arithmetic over rings like $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ can be implemented more efficiently in modern hardware than arithmetic over $\mathbb{F}_p$, which requires a software implementation for reduction modulo p. Also, non-arithmetic operations like comparison and truncation become simpler and more efficient in this setting [DEF+19,MR18].

1.1 Our Contributions

In this work we develop efficient protocols over $\mathbb{Z}_{2^k}$ by presenting a generic compiler that transforms any passively secure protocol for computation over $\mathbb{Z}_{2^{k+s}}$ that is only vulnerable to additive attacks, to a protocol over the ring $\mathbb{Z}_{2^k}$ which is actively secure with abort and provides roughly s bits of statistical security.[5] Both the input and output protocols are secure in the honest majority setting. The amortized cost of our compiler per multiplication gate is just *two invocations* of the passively secure protocol over the ring $\mathbb{Z}_{2^{k+s}}$. Furthermore, our compiler preserves the important property shared by some honest-majority multiplication protocols, which is that dot-products on shared vectors have the same communication cost as one single multiplication. This is crucial for many applications like secure array indexing [BKY19], or even more importantly applications relying on matrix arithmetic like SVMs or neural networks, as shown for example in [MR18].

We apply our compiler to two passively secure protocols over $\mathbb{Z}_{2^k}$ that are secure up to additive attack, and thus obtain two protocols.

The first protocol works for an arbitrary number of parties, and to the best of our knowledge we obtain the first actively secure protocol over $\mathbb{Z}_{2^k}$ that provides concrete efficiency in this setting. It is based on a version of Shamir secret sharing over rings [ACD+19]. We demonstrate the efficiency with an implementation, showing that looking only on the online computation, we are able to process a circuit of 1 million multiplications in depth-20 over the ring $\mathbb{Z}_{2^{64}}$ in overall 5 seconds (with 3 parties in a LAN networok), from which only 240ms are spent on the online computation. Because the size of the shares of our secret-sharing scheme is $k \log n$ bits, each party needs to communicate $14(k+s)\log n$ bits per multiplication gate. We thus obtain quasi-linear communication complexity in the number of parties.

The second protocol works in the three-party setting and is highly efficient. It is based on a protocol that uses replicated secret sharing and which is known to be very efficient [AFL+16]. Our compiled protocol for the three-party case requires each party to send just two elements in the ring $\mathbb{Z}_{2^{k+s}}$ per multiplication gate, i.e., communicating $2(k+s)$ bits per party. Furthermore, this protocol is the *first* actively secure three-party protocol over rings with the property that arbitrarily long secure dot products can be computed at the communication cost of one single multiplication, which, as we already argued, is an essential property for many applications. This is not the case for previous protocols like [EKO+19,FLNW17], which make use of Beaver-based preprocessing to achieve active security, do not achieve this important property.

We compare both protocols to other works, both theoretically and empirically. For the three-party case, our protocol yields the lowest overall bandwidth per multiplication gate for large rings to this date, improving upon the previous best result of [EKO+19]. Our implementation shows that million multiplications (with

[5] Although our protocols are statistically secure in principle, some efficient instantiations might make use of computational assumptions.

the same circuit as above) over the ring $\mathbb{Z}_{2^{64}}$ can be processed in 400ms in a LAN network.

Furthermore, our implementation of the first protocol includes an implementation of the secret-sharing scheme over $\mathbb{Z}_{2^k}$ via Galois rings of [ACD+19], which may have applications going beyond MPC. We demonstrate the practical viability of these techniques.

Our proof-of-concept implementation shows also that our compiled protocols perform well with respect to their field counterpart [CGH+18], which illustrates the benefits of working over the ring $\mathbb{Z}_{2^k}$ in terms of concrete efficiency. Specifically, our experiments show that although we double the amount of sent data, we provide in most cases faster computations.

1.2 Overview of our Techniques

The starting point of our work is the general compiler by Chida et al. [CGH+18], which achieves a similar result to ours, but over fields. In that work, the authors show that passively secure honest majority protocols over fields, which enjoy the additional property of being secure up to additive attacks, can be compiled to achieve active security without a noticeable loss in efficiency. The authors show several instantiations of their compiler, based on the observation from [GIP+14] that many existing passively secure protocols are already secure up to additive attacks.

Ideally, one would try to apply the results from [CGH+18] to compiler protocols over $\mathbb{Z}_{2^k}$. However, their techniques do not extend directly to the ring setting, mainly because the security of their compiler is based on the property that the code $x \to (x, r \cdot x)$ for a random r is resilient to additive attacks that are independent of x and r, and this does not hold for rings like $\mathbb{Z}_{2^k}$.

Our main observation is that a similar issue appears in the dishonest majority scenario, in which a MAC scheme is required in order to enforce correct behavior from the parties when reconstructing. More precisely, the MAC scheme used in SPDZ-like protocols for dishonest majority relies on the same property as above, and its generalization to the ring setting was not clear until the work of [CDE+18], where the authors showed how to generalize this authentication scheme to $\mathbb{Z}_{2^k}$, using a novel idea of extending the algebraic structure from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$ to provide some "extra room" for authentication.

At a very high level, our compiler is obtained by following the template from [CGH+18], using the "SPD$\mathbb{Z}_{2^k}$ trick" from [CDE+18] for the underlying AMD code, and it is described and analyzed in Section 4. However, extending the code $(x, r \cdot x)$ to $\mathbb{Z}_{2^k}$ is not the only critical piece of the compiler that fails when ported to the ring setting. For example, one of the critical steps in the compiler is to check whether a given shared value is 0, without revealing anything else about it (in particular, if the underlying value is not 0, nothing about it can be revealed). Over the field, this is done by multiplying the shared value by a secret random element, and opening this value. Clearly, with high probability, the opened value is zero if and only if the original value was zero, and if this is not the case then the opened value will look random.

Unfortunately, the check above does not work over $\mathbb{Z}_{2^k}$ (for example, if the original value is even and non-zero, then the opened value is even, which reveals the parity, with probability 1/2, of the original value). In this work we take a different path and show in Section 3.3 a novel check against zero that works over rings, which may be of independent interest. In a nutshell, our check works by bit-decomposing the shared value and checking that each bit is zero, which can be achieved by checking that the OR of the bits is zero. This idea stems from [DEF+19], where it was introduced in the dishonest majority setting for the particular type of secret sharing scheme used in the SPD$\mathbb{Z}_{2^k}$ protocol [CDE+18]. Here we extend this approach to the case of an arbitrary honest-majority secret sharing scheme that satisfies the conditions from Section 2.1. The resulting check is clearly more expensive than the one over fields, but since this is used only once in any execution of the protocol, this cost is amortized away.

Similarly to [CGH+18], we apply our compiler to two passively secure protocols: one based on replicated secret sharing and one based on Shamir secret sharing. For the first protocol, we simply observe in Section 5 that the proof presented in [LN17], which shows that replicated secret sharing over fields is secure up to additive attacks, also holds over $\mathbb{Z}_{2^k}$. This is natural since this scheme only uses the additive structure of the field, which is the same as $\mathbb{Z}_{2^k}$.

Now, for the second protocol, we rely on the recent work of [ACD+19] that extends Shamir secret sharing to the ring setting. In that work, the authors construct a protocol for the $t < n/2$ setting based on the protocol from [BTH06]. Since the goal of [ACD+19] is to construct a protocol with guaranteed output delivery, their techniques are quite complex and the concrete efficiency is not so clear. In this work we are interested in passively secure protocols with security up to additive attacks, so instead of directly compiling the protocol from [ACD+19], we use their core ideas together with techniques from [DN07] to develop in Section 6 a passively secure protocol over $\mathbb{Z}_{2^k}$ that is secure up to additive attacks.

Although the construction of the protocol is relatively straightforward given the tools presented in [ACD+19], the main complications arise in proving that this protocol is secure up to additive attacks, given that the protocol uses a Galois ring extension to be able to use polynomial interpolation and therefore, in principle, the adversary may inject non-additive errors by using elements that do not lie in the base ring, as shown in [ACD+19]. To remedy this issue, we tweak the basic "DN07" multiplication protocol so that it is ensured that the only possible attack by an active adversary is an additive attack over the base ring. We do this by a simple but novel approach to Shamir-based computation over an extension field/ring, which consists of sending one single base field/ring element to the "king", instead of a full field/ring extension element.

1.3 Related Work

In this section, we compare our protocol to the best concrete-efficient protocols for arithmetic computation over rings currently known. The only previous general compiler with concrete efficiency in this setting, to the best of our knowledge, is the compiler of [DOS18], which was improved recently by [EKO+19]. However,

their compiler does not preserve the threshold when moving from passive to active security. Thus, our compiler has a real qualitative advantage over thiers. In addition, in [DOS18] and [EKO+19] the compiler was instantiated for the 3-party case only.

The only concrete-efficient protocol for arithmetic computation over rings which works for *any* number of parties is the so-called "SPDZ2k" protocol [CDE+18] which was proven to be practical in [DEF+19]. This protocol is for the dishonest majority and thus requires the use of much heavier machinery, which makes it orders of magnitudes slower than ours. However, they deal with a more complicated setting and provide stronger security.

In the three-party setting with one corruption, there are several works which provide high efficiency for arithmetic computations over rings. The Sharemined protocol [BLW08] is being used to solve real-world problems but provides only semi-honest security. The actively secure protocol of [FLNW17], which was optimized and implemented in [ABF+17], is based on the "cut–and–choose" approach. This protocol requires each party to send 7 ring elements per multiplication gate. The advantage of their approach is that this amount stays the same also when working over small rings (e.g., boolean circuits). Thus, while we achieve lower bandwidth for large rings such as $\mathbb{Z}_{2^{32}}$ and $\mathbb{Z}_{2^{64}}$, their protocol will be favorable when working over small rings. The protocol of [CCPS19] has a slightly overall higher bandwidth than [ABF+17], but focuses on minimizing online (input-dependent) cost. Indeed, the online communication cost of [CCPS19] per party is just 4/3 ring elements per multiplication gate. Finally, the actively secure three-party protocol of [EKO+19] is the closest to our protocol in the sense that they also focus on efficiency for large rings. The protocol of [EKO+19] has two variants, with pre-processing and post-processing, which differs in where the computational bottleneck lies. The overall communication per multiplication gate of their protocol is $3(k+s)$ bits sent by each party, which is higher than ours by $(k+s)$ bits. We provide a detailed empirical comparison with [EKO+19] in Section 7.

We finally remark that in [GRW18], a protocol for 4-party and one corruption was presented, which requires each party to send just 1.5 ring elements for each multiplication gate. However, the protocol applies for this specific setting only and was not implemented.

2 Preliminaries and Definitions

Notation. Let $P_1, \ldots, P_n$ denote the n parties participating in the computation, and let t denote the number of corrupted parties. In this work, we assume an honest majority, and thus $t < \frac{n}{2}$. Throughout the paper, we use H to denote the subset of honest parties and $\mathcal{C}$ to denote the subset of corrupted parties. We use $[n]$ to denote the set $\{1, \ldots, n\}$. $\mathbb{Z}_M$ denotes the ring of integers modulo M, and the congruence $x \equiv y \bmod 2^\ell$ is denoted by $x \equiv_\ell y$.

2.1 Secret Sharing

Let ℓ be a positive integer. A t-out-of-n secret sharing scheme over $\mathbb{Z}_{2^\ell}$ distributes an input $x \in \mathbb{Z}_{2^\ell}$ among the n parties $P_1, \ldots, P_n$, giving *shares* to each one of them in such a way that any subset of at least $t+1$ parties can reconstruct x from their shares, but any subset of at most t parties cannot learn anything about x from their shares. We formalize this notion as follows.

Definition 2.1. *A t-out-of-n linear secret sharing scheme over $\mathbb{Z}_{2^\ell}$ is a pair of interactive procedures* share *and* open *that satisfy the following properties.*

Share-Distribution Procedure. $\mathsf{share}(x)$ *is as randomized efficiently-computable procedure that generates n shares $(x_1, \ldots, x_n)$ of $x \in \mathbb{Z}_{2^\ell}$, where $x_i \in (Z_{2^\ell})^m$ is intended for party P_i.*[6]
Given a subset $J \subseteq [n]$, we denote $[\![x]\!]_\ell^J = \{x_i\}_{i\in J}$, and if $J = [n]$ we simply write $[\![x]\!]_\ell^{[n]} = [\![x]\!]_\ell$. Furthermore, if ℓ is clear from the context we may omit the subscript ℓ.

Share-Distribution From Given Shares. *The* share *algorithm above may also take as input, in addition to $x \in \mathbb{Z}_{2^\ell}$, a set of shares $\{x_i\}_{i\in J}$ for $J \subseteq [n]$ with $|J| \leq t$ so that its output $[\![x]\!] = \mathsf{share}(x, \{x_i\}_{i\in J})$ satisfies $[\![x]\!] = (x'_1, \ldots, x'_n)$, with $x'_i = x_i$ for $i \in J$.*
We assume that if $|J| = t$, then $[\![x]\!]^J$ together with v determine deterministically *all the remaining shares. This also means that any $t+1$ shares fully determine all shares.*

Privacy. *For any $J \subseteq [n]$ with $|J| \leq t$, the mutual information between $\{x_i\}_{i\in J}$ and x is zero, where $\mathsf{share}(x) = \{x_i\}_{i\in[n]}$.*

Reconstruction. open *is an efficiently-computable deterministic procedure such that $\mathsf{open}([\![x]\!]^J) = x$ or $\perp$ for every $J \subseteq [n]$ with $|J| > t$.*
In particular, the procedure open *outputs a special symbol $\perp$ whenever it is called on an input $[\![x]\!]$ which is not* correct *as defined below in Definition 2.2. The procedure may take an extra common index $i \in [n]$ as in $\mathsf{open}([\![x]\!], i)$, and in such a case, the output is obtained only by P_i.*

Shares of a Constant. *There exists a deterministic procedure* sharecons *such that, on input $x \in \mathbb{Z}_{2^\ell}$, produces $\{x_i\}_{i\in[n]}$ such that $\mathsf{open}(\{x_i\}_{i\in[n]}) = x$. Furthermore, we assume that all the entries of $\{x_i\}_{i\in[n]}$ are either equal to 0 or equal to x.*

Homomorphism. *Given shares $[\![x]\!]$, $[\![y]\!]$, point-wise addition of these shares yields shares of $x + y \bmod 2^\ell$. We denote this operation by $[\![x]\!] + [\![y]\!]$.*[7]

We also assume the following, non-standard properties:

[6] Notice that the shares x_i do not necessarily live in $\mathbb{Z}_{2^\ell}$. For example, for replicated secret-sharing scheme these shares belong to $\mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$, and for our instantiation of Shamir secret sharing over rings these shares belong to $\mathbb{Z}_{2^\ell}^{\log n}$.

[7] Notice that, given $[\![x]\!]$ and $\alpha \in \mathbb{Z}_{2^\ell}$, one can compute shares of $x + \alpha \bmod 2^\ell$ by calling sharecons on input α and then adding the shares point-wise.

Modular Reduction. *We assume that the* open *procedure is compatible with modular reduction, meaning that for any* $0 \leq \ell' \leq \ell$ *and any* $x \in \mathbb{Z}_{2^\ell}$*, reducing each share in* $[\![x]\!]_\ell$ *modulo* $2^{\ell'}$ *yields shares* $[\![x \bmod 2^{\ell'}]\!]_{\ell'}$*. We denote this by* $[\![x]\!]_\ell \to [\![x]\!]_{\ell'}$*.*

Multiplication by $1/2$**.** *Given a shared value* $[\![x]\!]_\ell$*, we assume if all the shares are even then shifting these shares to the right yields shares* $[\![x']\!]_{\ell-1}$*, where* $x' = x/2$*.*[8]

Throughout the entire paper, we set the threshold for the secret-sharing scheme to be $\lfloor \frac{n-1}{2} \rfloor$, and we denote by t the number of corrupted parties. Since we assume an honest majority, it holds that $t < n/2$ and so the corrupted parties can learn nothing about a shared secret. This also means that the shares of the honest parties always fully determine the shares of the corrupted parties (this follows from the "share-distribution from given shares" property stated above since $H > n/2 > t$). We will use this property frequently in our functionalities.

Now we define what it means for the parties to have *correct* shares of some value. Let J be a subset of honest parties of size $t+1$, and denote by $\mathsf{val}([\![v]\!])_J$ the value obtained by these parties after running the open protocol, where no corrupted parties or additional honest parties participate, i.e. $\mathsf{open}([\![v]\!]^J)$. Note that $\mathsf{val}([\![v]\!])_J$ may equal $\perp$ and in this case we say that the shares held by the honest parties are not valid. Informally, a secret sharing is correct if every subset of $t+1$ honest parties reconstruct the same value (which is not $\perp$). Formally:

Definition 2.2. *Let* $H \subseteq \{P_1, \ldots, P_n\}$ *denote the set of honest parties. A sharing* $[\![v]\!]$ *is* correct *if there exists a value* $v' \in \mathbb{F}$ $(v' \neq \perp)$ *such that for every* $J \subseteq H$ *with* $|J| = t+1$ *it holds that* $\mathsf{val}([\![v]\!])_J = v'$*.*

2.2 Security Definition

We use the standard definition of security based on the ideal/real model paradigm [Can00,Gol04], with security formalized for non-unanimous abort. This means that the adversary first receives the output, and then determines for each honest party whether they will receive abort or receive their correct output. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution [GL05]. For simplicity, we omit this step from the description of our protocols. Our protocol is cast in the synchronous model of communication, in which it is assumed that the parties share a common clock and protocols can be executed in rounds.

2.3 Secure Multiplication up to Additive Attacks [GIP15,GIP+14]

Our construction works by running a multiplication protocol (for multiplying two values that are shared among the parties) that is *not* fully secure in the

[8] If all the shares $[\![x]\!]_\ell$ are even then these shares may be written as $[\![x]\!]_\ell = 2 \cdot [\![y]\!]_\ell$, which, by the homomorphism property, are shares of $2 \cdot y$. Since these are shares of x as well, this shows that $x \equiv_\ell 2 \cdot y$, so x is even.

presence of a malicious adversary and then running a verification step that enables the honest parties to detect cheating. In order to do this, we start with a multiplication protocols with the property that the adversary's ability to cheat is limited to carrying out a so-called "additive attack" on the output. Formally, we say that a multiplication protocol is secure up to an additive attack if it realizes $\mathcal{F}_{\mathsf{mult}}$ defined in Functionality 2.3. This functionality receives input sharings $[\![x]\!]$ and $[\![y]\!]$ from the honest parties and an additive value d from the adversary, and outputs a sharing of $x \cdot y + d$. Since the corrupted parties can determine their own shares in the protocol, the functionality allows the adversary to provide the shares of the corrupted parties, but this reveals nothing about the shared value.

As we will discuss in the instantiations sections (Section 5 and 6), the requirements defined by this functionality can be met by some semi-honest multiplication protocols over $\mathbb{Z}_{2^\ell}$, namely replicated secret sharing and the more recent protocol of Cramer et al. [ACD+19], which is an extension of Shamir Secret Sharing to the setting of $\mathbb{Z}_{2^\ell}$. This will allow us to implement this functionality in a very efficient way.

FUNCTIONALITY 2.3 ($\mathcal{F}_{\mathsf{mult}}(\ell)$ - Secure Mult. Up To Additive Attack)

1. Upon receiving $[\![x]\!]_\ell^H$ and $[\![y]\!]_\ell^H$ from the honest parties, the ideal functionality $\mathcal{F}_{\mathsf{mult}}$ computes x, y and the corrupted parties shares $[\![x]\!]_\ell^{\mathcal{C}}$ and $[\![y]\!]_\ell^{\mathcal{C}}$.
2. $\mathcal{F}_{\mathsf{mult}}$ hands $[\![x]\!]_\ell^{\mathcal{C}}$ and $[\![y]\!]_\ell^{\mathcal{C}}$ to the ideal-model adversary/simulator $\mathcal{S}$.
3. Upon receiving d and $\{\alpha_i\}_{i|P_i \in \mathcal{C}}$ from $\mathcal{S}$, functionality $\mathcal{F}_{\mathsf{mult}}$ defines $z \equiv_\ell x \cdot y + d$ and $[\![z]\!]_\ell^{\mathcal{C}} = \{\alpha_i\}_{i|P_i \in \mathcal{C}}$. Then, it runs $\mathsf{share}(z, [\![z]\!]_\ell^{\mathcal{C}})$ to obtain a share z_j for each party P_j.
4. The ideal functionality $\mathcal{F}_{\mathsf{mult}}$ hands each honest party P_j its share z_j.

Efficient Sum of Products. In addition to the above, we define a functionality that allows the parties to securely compute the dot product of two vectors of shares, where the adversary is allowed to inject an additive error to the final output. This is described in Functionality 2.4. As in [CGH+18], we will show that the functionality can be realized at almost the same cost as $\mathcal{F}_{\mathsf{mult}}$.

FUNCTIONALITY 2.4 ($\mathcal{F}_{\mathsf{DotProduct}}(\ell)$ - Sum of Products Up To Add. Attack)

1. Upon receiving $\{[\![x_i]\!]_\ell^H\}_{i=1}^m$ and $\{[\![y_i]\!]_\ell^H\}_{i=1}^m$ from the honest parties, $\mathcal{F}_{\mathsf{DotProduct}}$ recovers x, y and computes the corrupt parties shares $\{[\![x_i]\!]_\ell^{\mathcal{C}}\}_{i=1}^m$ and $\{[\![y_i]\!]_\ell^{\mathcal{C}}\}_{i=1}^m$, and sends these shares to the ideal adversary $\mathcal{S}$.
2. Upon receiving d and $[\![z]\!]_\ell^{\mathcal{C}} = \{\alpha_i\}_{i|P_i \in \mathcal{C}}$ from $\mathcal{S}$, define $z \equiv_\ell d + \sum_{i=1}^m x_i y_i$.
3. Run $\mathsf{share}(z, [\![z]\!]_\ell^{\mathcal{C}})$ to obtain a share z^j for each P_j.
4. Return z^j to P_j.

3 Building Blocks and Sub-Protocols

Our compiler requires a series of building blocks in order to operate. These include generation of random shares and public coin-tossing, as well as broadcast. Furthermore, as mentioned in Section 1.2, a core step of our compiler is checking that a shared value is zero, leaking nothing more than this binary information. We define this functionality and instantiate it in Section 3.3. We stress that our presentation here is very general and it assumes nothing about the underlying secret sharing scheme beyond the properties stated in Section 2.1.

3.1 Basic Building Blocks

$\mathcal{F}_{\mathsf{rand}}$ - Generating Random Shares. We define the ideal functionality $\mathcal{F}_{\mathsf{rand}}$ to generate a sharing of a random value unknown to the parties. The functionality lets the adversary choose the corrupted parties' shares, which together with the random secret chosen by the functionality, are used to compute the shares of the honest parties.

The way to compute this functionality depends on the specific secret sharing scheme that is being used. For example, for the case of replicated secret sharing we consider the well-known method [AFL+16] that is based on distributing replicated keys for a PRF, which allows the parties to generate shares of random values without interaction. For the case of Shamir secret sharing (Section 6.1), we consider an instantiation which relies on super-invertible matrices [DN07] to achieve linear communication complexity, together with the "tensoring-trick" from [CCXY18,ACD+19] in order to instantiate such matrices efficiently.

$\mathcal{F}_{\mathsf{coin}}$ - Generating Random Coins. $\mathcal{F}_{\mathsf{coin}}(\ell)$ is an ideal functionality that chooses a random element from $\mathbb{Z}_{2^\ell}$ and hands it to all parties. A simple way to compute $\mathcal{F}_{\mathsf{coin}}$ is to use $\mathcal{F}_{\mathsf{rand}}$ to generate a random sharing and then open it. In the plain model, one can generate random coins by having each party (more precisely, it suffices for $t+1$ parties to do this) shares a random secret, which are then summed by the parites and opened to reveal the sum of the secrets. The fact that there is a least one honest party which shares a secret guarantees that the obtained value is uniformly distributed over the ring. On the other hand, the properties of the open procedure, guarantee that if the corrupted parties distributed an incorrect sharing, the honest parties will detect it and abort.

$\mathcal{F}_{\mathsf{bc}}$ - Broadcast with Abort Another essential primitive for our compiler is broadcast, in which a given party sends a message to all other parties, with the guarantee that all the honest parties agree on the same value. Furthermore, if the sender is honest, the agreed value is precisely the one that the sender sent. It is well-known that broadcast cannot be achived when $t \geq n/3$ without any trusted set-up [PSL80]. However, for our protocol, we need only a weaker notion of broadcast with abort, meaning that the adversary can cause the parites to abort (but not to output an incorrect message).

A simple way to compute $\mathcal{F}_{\text{bc}}$ is the well-known echo-broadcast protocol, where the parties echo the message they received and send it the other parties. Note that this protocol does not achieve unanimous abort and gives the adversary the ability to determine which of the honest parties will output the sent message and which will abort (see Section 2.2).

3.2 $\mathcal{F}_{\text{input}}$ – Secure Sharing of Inputs

In this section, we present our protocol for secure sharing of the parties' inputs. The protocol is the same as in [CGH+18] (and many prior works): for each input x belonging to a party P_j, the parties call $\mathcal{F}_{\mathsf{rand}}$ to generate a random sharing $[\![r]\!]$; denote the share held by P_i by r_i. Then, r is reconstructed to P_j, who broadcasts $x - r$ to all parties. Finally, each P_i outputs the share $[\![r + (x - r)]\!] = [\![x]\!]$. This is secure since $\mathcal{F}_{\mathsf{rand}}$ guarantees that the sharing of r is correct, which in turn guarantees that the sharing of x is correct (since adding $x - r$ is a local operation only). In order to ensure that P_j sends the same value $x - r$ to all parties, $\mathcal{F}_{\text{bc}}$ is used. As shown in [CGH+18], the above protocol securely computes with abort the ideal functionality for input sharing described in Functionality 3.1.

FUNCTIONALITY 3.1 ($\mathcal{F}_{\text{input}}(\ell)$- Sharing of Inputs)

1. Functionality $\mathcal{F}_{\text{input}}$ receives inputs $v_1, \ldots, v_M \in \mathbb{Z}_{2^\ell}$ from the parties. For every $i = 1, \ldots, M$, $\mathcal{F}_{\text{input}}$ also receives from $\mathcal{S}$ the shares $[\![v_i]\!]_\ell^C$ of the corrupted parties for the ith input.
2. For every $i = 1, \ldots, M$, $\mathcal{F}_{\text{input}}$ computes all shares $(v_i^1, \ldots, v_i^n) = \mathsf{share}(v_i, [\![v_i]\!]_\ell^C)$. For every $j = 1, \ldots, n$, $\mathcal{F}_{\text{input}}$ sends P_j its output shares $(v_1^j, \ldots, v_M^j)$.

3.3 $\mathcal{F}_{\text{CheckZero}}$ - Checking Equality to 0

A key component of our compiler is a protocol for checking whether a given sharing is a sharing of the value 0, without revealing any extra information on the shared value.

More precisely, let $v \in \mathbb{Z}_{2^\ell}$, and suppose that the parties hold a sharing $[\![v]\!]_\ell$. The parties want to check whether $v \equiv_\ell 0$, while guaranteeing that nothing is learned about $v \bmod 2^\ell$ if this is not the case. This is required due to the way we will use this check in our protocol: an adversary can make v depend on the inputs of honest parties, so if the parties simply open v and check that it is zero then the adversary gets to learn a function of the inputs.

A simple way to approach this problem when working over a field is sampling a random multiplicative mask $[\![r]\!]$, multiply $[\![r \cdot v]\!] = [\![r]\!] \cdot [\![v]\!]$, open $r \cdot v$ and check that it is equal to zero. Clearly, since r is random then $r \cdot v$ looks also random if $v \neq 0$. However, this technique does not work over the ring $\mathbb{Z}_{2^\ell}$: for example, if

v is a non-zero even number then $r \cdot v$ is always even, which reveals too much about v.

In this section we present a generic protocol to solve the problem of checking equality of zero over the ring, which is more expensive and complicated than the protocol over fields described above. Fortunately, this check is only called *once* in a full execution of the main protocol and so the complexity of this technique is amortized away. Furthermore, the check we present here is generic and does not assume anything about the underlying secret sharing scheme, but for some specific instantiations one can get a much more efficient solution. For example, we show in Sections 5 and 6 how to instantiate this check efficiently for the case of replicated secret sharing and shamir secret sharing, respectively.

The functionality we want to realize ,$\mathcal{F}_{\text{CheckZero}}$, is described formally in Functionality 3.2. $\mathcal{F}_{\text{CheckZero}}$ determines the value of the secret v based on the honest parties' shares and then it sends accept or reject to the parties. In addition, it computes the corrupted parties' shares of v from the honest parties' shares and hand them to the ideal world adversary $\mathcal{S}$.

FUNCTIONALITY 3.2 ($\mathcal{F}_{\text{CheckZero}}(\ell)$ – Checking Equality to 0)
The ideal functionality $\mathcal{F}_{\text{CheckZero}}$ receives $[\![v]\!]_\ell^H$ from the honest parties and uses them to compute v and $[\![v]\!]_\ell^C$, the shares of v of the corrupt parties. Then, $\mathcal{F}_{\text{CheckZero}}$ hands $[\![v]\!]_\ell^C$ to the simulator $\mathcal{S}$.
The output is determined by $\mathcal{F}_{\text{CheckZero}}$ as follows:

- If $v \equiv_\ell 0$, then $\mathcal{F}_{\text{CheckZero}}$ sends accept to the honest parties and $\mathcal{S}$.
- If $v \not\equiv_\ell 0$, then it sends reject to the honest parties and $\mathcal{S}$.

The simple observation behind our protocol to compute $\mathcal{F}_{\text{CheckZero}}$ (which follows the idea of [Cd10,DEF+19]) is that v is zero if and only if $v + r \equiv_\ell r$ for every $r \in \mathbb{Z}_{2^\ell}$. Moreover, if r is secret, the parties can open $c = v + r$ without leaking v. Then, to check that $v + r \equiv_\ell r$, the parties can check that the bit- representation of the two values is identical. Since $c = v + r$ is made public, each party can locally decompose it to bits for this check. In addition, the parties choose the sharing of the secret r by first computing random shared bits $[\![r_0]\!]_\ell, \ldots, [\![r_{\ell-1}]\!]_\ell$ (note that here each r_k is a bit which is shared over the ring $\mathbb{Z}_{2^\ell}$) which are then locally composed to obtain $[\![r]\!]_\ell$. Thus, the bit representation of r is shared between the parties and can be used for the check. To complete the construction of the protocol, we need to solve two issues. First, we need a protocol to produce random shared bits. We thus define the ideal functionality $\mathcal{F}_{\text{randBit}}$ which is identical to $\mathcal{F}_{\mathsf{rand}}$ except that the random value is chosen by the functionality as a bit. The protocol to compute $\mathcal{F}_{\text{randBit}}$, which we present below, builds upon $\mathcal{F}_{\mathsf{rand}}$ and an ideal functionality $\mathcal{F}_{\text{CorrectMult}}$ which performs correct multiplication over shared values (as oppose to $\mathcal{F}_{\mathsf{mult}}$ which allows the adversary to change the output). We explain how to compute $\mathcal{F}_{\text{CorrectMult}}$ below. The second issue is how to check that all bits of $v + r$ and the shared r are

identical. This is done by computing a circuit which XOR each bit of $v + r$ with its corresponding bit of r and then outputs the OR of the xored bits. If all bits are indentical then the result should be 0. To compute the circuit, the parties once again use the $\mathcal{F}_{\text{CorrectMult}}$ functionality.

The general protocol to compute $\mathcal{F}_{\text{CheckZero}}$ is described in Protocol 3.8. We begin, however, by presenting our protocols to compute $\mathcal{F}_{\text{CorrectMult}}$ and $\mathcal{F}_{\text{randBit}}$. We stress again that $\mathcal{F}_{\text{CorrectMult}}$ is more difficult to achieve than $\mathcal{F}_{\mathsf{mult}}$ and hence the cost is much higher. However, we call $\mathcal{F}_{\text{CorrectMult}}$ only a constant number of times during the execution, and thus the overall overhead is very reasonable.

Computing $\mathcal{F}_{\text{CorrectMult}}$ via Sacrificing. As explained above, $\mathcal{F}_{\text{CorrectMult}}$ is an ideal functionality which receives shares of two inputs from the honest parties and a set of shares from the corrupted parties, to hand the honest parties random shares of the input's multiplication, which are chosen given the shares that were received from the corrupted parties. Our protocol to compute this is based on a technique known as "sacrificing". The idea is to generate correct random multiplication triples, which are then consumed to multiply the inputs. This is done by calling $\mathcal{F}_{\mathsf{rand}}$ three times to obtain random shares $[\![a]\!], [\![b]\!], [\![a']\!]$, calling $\mathcal{F}_{\mathsf{mult}}$ twice to obtain $[\![a \cdot b]\!]$ and $[\![a' \cdot b]\!]$, and using one triple to check the correctness of the other. Some modifications are needed in order to make this work over the ring $\mathbb{Z}_{2^\ell}$ for which we use the "SPDZ2k trick" from [CDE+18]. This requires us to perform the check over the ring $\mathbb{Z}_{2^{\ell+s}}$, thereby achieving a statistical error of 2^{-s}. The construction is presented in detail in Protocol 3.3.

Note that the protocol can be divided into two stages: an offline phase where the multiplication triple is generated, and an online phase where the triple is used to compute the product of the given shares. Thus, an efficient implementation would batch all the preprocessing together, and then proceed to consume these triples when the actual multiplication is required.

We remark that other approaches to produce random triples, such as "cut–and–choose", would work here as well. However, the "cut–and–choose" method becomes efficient only when many triples are being generated together—much more than what is needed by our protocol (for example, in [FLNW17], to achieve good parameters for the "cut–and–choose" process which yield low bandwidth, 2^{20} triples are generated together). Thus, the sacrificing approach is favorable in our setting.

PROTOCOL 3.3 (Correct Multiplication)

– **Inputs:** Two shares $[\![x]\!]_\ell$ and $[\![y]\!]_\ell$ to be multiplied.
– **The protocol:**
1. Generate a multiplication triple via sacrificing.
   (a) The parties call $\mathcal{F}_{\mathsf{rand}}(\ell + s)$ three times to obtain sharings $[\![a]\!]_{\ell+s}, [\![a']\!]_{\ell+s}, [\![b]\!]_{\ell+s}$.
   (b) The parties call $\mathcal{F}_{\mathsf{mult}}(\ell + s)$ on input $[\![a]\!]_{\ell+s}$ and $[\![b]\!]_{\ell+s}$ to obtain shares $[\![c]\!]_{\ell+s}$, and on input $[\![a']\!]_{\ell+s}$ and $[\![b]\!]_{\ell+s}$ to obtain shares $[\![c']\!]_{\ell+s}$.
   (c) The parties call $\mathcal{F}_{\mathsf{coin}}(s)$ to obtain a random element $r \in \mathbb{Z}_{2^s}$.
   (d) The parties execute $\mathsf{open}(r \cdot [\![a]\!]_{\ell+s} - [\![a']\!]_{\ell+s}) = a''$.
   (e) The parties execute $\mathsf{open}(a'' \cdot [\![b]\!]_{\ell+s} - r \cdot [\![c]\!]_{\ell+s} + [\![c']\!]_{\ell+s}) = w$ and check that $w \equiv_{\ell+s} 0$.
   (f) If the check in the previous step has failed, the parties abort. Otherwise they compute $[\![\pi]\!]_{\ell+s} \to [\![\pi]\!]_\ell$ for $\pi \in \{a, b, c\}$, take $([\![a]\!]_\ell, [\![b]\!]_\ell, [\![c]\!]_\ell)$ as a valid triple and continue to the next step.
2. Use the generated triple to multiply the input shares.
   (a) The parties execute $\mathsf{open}([\![x]\!]_\ell - [\![a]\!]_\ell) = u$ and $\mathsf{open}([\![y]\!]_\ell - [\![b]\!]_\ell) = v$.
   (b) The parties locally compute $[\![z]\!]_\ell = [\![c]\!]_\ell + u \cdot [\![b]\!]_\ell + v \cdot [\![a]\!]_\ell + u \cdot v$.
– **Outputs:** The parties output the shares $[\![z]\!]_\ell$.

To argue the security of Protocol 3.3, we use the following lemma which shows that sacrificing leads to a correct triple with high probability. This is the same argument as the one presented in [CDE+18].

Lemma 3.4. *If the check at the end of the first step in Protocol 3.3 passes, then the additive error $d \in \mathbb{Z}_{2^{\ell+s}}$ that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{mult}}$ is zero modulo 2^ℓ with probability at least $1 - 2^{-s}$.*

Proof: Since $\mathcal{F}_{\mathsf{mult}}$ is used in the first step, we have that $c = a \cdot b + d$ and $c' = a' \cdot b + d'$, where $d, d' \in \mathbb{Z}_{2^{\ell+s}}$ are the additive attacks chosen by the adversary in the first and second call to $\mathcal{F}_{\mathsf{mult}}$ respectively. It follows that $a'' \cdot b - r \cdot c + c' \equiv_{\ell+s} d' - r \cdot d$. Hence, if 2^v is the largest power of 2 dividing d, it holds that if $w \equiv_{\ell+s} 0$ then $\frac{r}{2^v} \equiv_{\ell+s-v} \left(\frac{d}{2^v}\right)^{-1} \frac{d'}{2^v}$, which holds with probability at most $2^{-(\ell+s-v)}$. If $d \not\equiv_\ell 0$, then $v > \ell$ and therefore this probability is upper bounded by 2^{-s}, which concludes the proof. ∎

With this lemma at hand we proceed to prove the security of Protocol 3.3. The key intuition is that the preprocessed triple is correct with high probability, and since the open procedure is guaranteed to yield the correct value, it is ensured that final linear combination gives the right product.

Proposition 3.5. *Protocol 3.3 securely computes functionality $\mathcal{F}_{\text{CorrectMult}}$ with abort and with statistical error 2^{-s} in the $(\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model in the presence of malicious adversaries controlling $t < n/2$ parties.*

The proof appears in Section A.1.

$\mathcal{F}_{\text{randBit}}$ - Generating Random Shared Bits. We now present our protocol to generate random shared bits. As discussed above, the protocol realizes the functionality $\mathcal{F}_{\text{randBit}}$, which is defined similarly to $\mathcal{F}_{\mathsf{rand}}$: it receives a set of shares from the adversary controlling the corrupted parties, to then choose a random bit and compute the honest parties's shares, given that the corrupted parties' shares are fixed. We stress that the resulted sharing is a sharing of a bit over the ring $\mathbb{Z}_{2^\ell}$.

We instantiate this functionality essentially by showing that the bit-generation procedure from [DEF+19], which is presented in the setting of SPDZ-type of shares, also extends to more general secret-sharing schemes. The main tool needed here is the "Multiplication by 1/2" property presented in Section 2.1, which states that parties can locally divide their shares of a secret $x \bmod 2^\ell$ by 2 to obtain shares of $x/2 \bmod 2^{\ell-1}$, as long as the shares and the secret are even.

Proposition 3.6. *Protocol 3.7 securely computes functionality $\mathcal{F}_{\text{randBit}}$ with abort in the $(\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\text{CorrectMult}})$-hybrid model in the presence of malicious adversaries controlling $t < n/2$ parties.*

The proof appears in Section A.2.

PROTOCOL 3.7 (Random Shared Bits Generation)

– **The protocol:**
1. The parties call $\mathcal{F}_{\mathsf{rand}}(\ell+2)$ to obtain a shared value $[\![r]\!]_{\ell+2}$. Then, the parties set $[\![a]\!]_{\ell+2} = 2 \cdot [\![r]\!]_{\ell+2} + 1$.
2. The parties call $\mathcal{F}_{\text{CorrectMult}}(\ell+2)$ on input $[\![a]\!]_{\ell+2}$ and $[\![a]\!]_{\ell+2}$ to obtain shares $[\![c]\!]_{\ell+2} = [\![a^2]\!]_{\ell+2}$. Then, they run $\mathsf{open}([\![c]\!]_{\ell+2})$ to obtain c.
3. The parties compute $[\![d]\!]_{\ell+2} = \sqrt{c}^{-1} \cdot [\![a]\!]_{\ell+2}$, where $\sqrt{c}$ is a fixed square root of c modulo $2^{\ell+2}$, and the inverse is taken modulo $2^{\ell+2}$.
4. The parties locally convert $[\![d]\!]_{\ell+2} \to [\![d]\!]_{\ell+1}$, and compute $[\![b']\!]_{\ell+1} = [\![d]\!]_{\ell+1} + 1$.
5. The parties locally shift their shares of b' one position to the right to obtain shares $[\![b]\!]_\ell$, where $b \equiv_\ell \frac{b'}{2}$.

– **Outputs:** The parties output $[\![b]\!]_\ell$.

Check Equality to Zero. We are now ready to formally present our check-to-zero protocol which is described in Protocol 3.8. As explained at the beginning of the section, the idea behind the protocol is to check that the bit representation of $v + r$ is identical to the bit representation of r, where r is sampled randomly from $\mathbb{Z}_{2^\ell}$.

PROTOCOL 3.8 (Checking Equality to 0)

– **Inputs:** The parties hold a sharing $[\![v]\!]_\ell$.
– **The protocol:**
1. The parties call $\mathcal{F}_{\text{randBit}}$ to get ℓ random shared bits $[\![r_0]\!]_\ell, \ldots, [\![r_{\ell-1}]\!]_\ell$.
2. The parties bit-decompose v:
   (a) The parties compute $[\![r]\!]_\ell = \sum_{i=0}^{\ell-1} 2^i \cdot [\![r_i]\!]_\ell$.
   (b) The parties call $c = \mathsf{open}([\![v]\!]_\ell + [\![r]\!]_\ell)$ and bit-decompose this value as $(c_0, \ldots, c_{\ell-1})$.
   (c) The parties locally convert $[\![r_i]\!]_\ell \to [\![r_i]\!]_1$ for $i = 1, \ldots, \ell - 1$.
3. The parties check that all the bits of $v \bmod 2^\ell$ are zero:
   (a) The parties use $\mathcal{F}_{\text{CorrectMult}}(1)$ to compute $\bigvee_{i=0}^{\ell-1}([\![r_i]\!]_1 \oplus c_i)$ and open this result.
   (b) If the opened value above is equal to 0 then the parties output accept. Otherwise they output reject.

Proposition 3.9. *Protocol 3.8 securely computes $\mathcal{F}_{\text{CheckZero}}$ with abort in the $(\mathcal{F}_{\text{randBit}}, \mathcal{F}_{\text{CorrectMult}})$-hybrid model in the presence of malicious adversaries who control $t < n/2$ parties.*

The proof appears in Section A.3.

Efficiency analysis. The main bottleneck of the above protocol is the costly $\mathcal{F}_{\text{CorrectMult}}$ functionality. Note that it is called ℓ times in Protocol 3.8 (for computing $\bigvee_{i=0}^{\ell-1}(r_i \oplus c_i)$) and once each time $\mathcal{F}_{\text{randBit}}$ is called. Thus, overall, it is called 2ℓ times. For example, for the ring $\mathbb{Z}_{2^{64}}$, this translates to 128 calls to $\mathcal{F}_{\text{CorrectMult}}$. Since $\mathcal{F}_{\text{CheckZero}}$ is called exactly once in our main protocol for computing a circuit, the overhead is not significant.

4 The Main Protocol for Rings

In this section, we present our construction to compute arithmetic circuits over the ring $\mathbb{Z}_{2^k}$. A formal description appears in Protocol 4.1. Our protocol follows the paradigm of [CGH+18] which works as follows. Each input to the circuit is randomized using a random sharing $[\![r]\!]$. This is done by taking each input $[\![v]\!]$ and multiply it with $[\![r]\!]$. Once the parties hold a pair of sharings on each input wire $([\![v]\!], [\![r \cdot v]\!])$, the parties go over the circuit while maintaining this invariant. For linear gates this can be done locally by each party due to the homomorphism property of the secret sharing scheme. For multiplication gates, with two inputs with sharings $([\![x]\!], [\![r \cdot x]\!])$ and $([\![y]\!], [\![r \cdot y]\!])$, the parties run a multiplication protocol twice, to multiply $[\![x]\!]$ and $[\![y]\!]$ and to multiply $[\![r \cdot x]\!]$ and $[\![y]\!]$. To carry out all the above multiplications, the parties use the functionality $\mathcal{F}_{\mathsf{mult}}$, which only guarantees security up to additive attack (and thus can be instantiated by highly-efficient protocols as we will see in Section 5 and Section 6).

PROTOCOL 4.1 (Computing Arithmetic Circuits Over the Ring $\mathbb{Z}_{2^k}$)

Inputs: Each party P_j ($j \in \{1,\ldots,n\}$) holds an input $x_j \in \mathbb{Z}_{2^k}^L$.

Auxiliary Input: The parties hold the description of an arithmetic circuit C over $\mathbb{Z}_{2^k}$ that computes f on inputs of length $M = L \cdot n$. Let N be the number of multiplication gates in C. In addition, the parties hold a parameter $s \in \mathbb{N}$.

The protocol:

1. *Secret sharing the inputs:*
   (a) For each input x_j held by party P_j, party P_j represent it as an element of $\mathbb{Z}_{2^{k+s}}^L$ and sends x_j to $\mathcal{F}_{\text{input}}(k+s)$.
   (b) Each party P_j records its vector of shares $(x_1^j,\ldots,x_M^j)$ of all inputs, as received from $\mathcal{F}_{\text{input}}(k+s)$. If a party received $\perp$ from $\mathcal{F}_{\text{input}}$, then it sends abort to the other parties and halts.
2. *Generate randomizing shares:* The parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![r]\!]_{k+s}$, where $r \in_R \mathbb{Z}_{2^{k+s}}$.
3. *Randomization of inputs:* For each input wire sharing $[\![v_m]\!]_{k+s}$ (where $m \in \{1,\ldots,M\}$) the parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![r]\!]_{k+s}$ to receive $[\![r \cdot v_m]\!]_{k+s}$.
4. *Circuit emulation:* The parties traverse over the circuit in topological order. For each gate G_ℓ the parties work as follows:
   – *G_ℓ is an addition gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively, the parties locally compute $([\![x+y]\!]_{k+s}, [\![r \cdot (x+y)]\!]_{k+s})$.
   – *G_ℓ is a multiplication-by-a-constant gate:* Given a constant $a \in \mathbb{Z}_{2^k}$ and tuple $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ on the input wire, the parties locally compute $([\![a \cdot x]\!]_{k+s}, [\![r \cdot (a \cdot x)]\!]_{k+s})$.
   – *G_ℓ is a multiplication gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively:
     (a) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![x \cdot y]\!]_{k+s}$.
     (b) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![r \cdot x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![r \cdot x \cdot y]\!]_{k+s}$.
5. *Verification stage:* Let $\{([\![z_i]\!]_{k+s}, [\![r \cdot z_i]\!]_{k+s})\}_{i=1}^N$ be the tuples on the output wires of all multiplication gates and let $\{[\![v_m]\!]_{k+s}, [\![r \cdot v_m]\!]_{k+s}\}_{m=1}^M$ be the tuples on the input wires of the circuit.
   (a) For $m = 1,\ldots,M$, the parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![\beta_m]\!]_{k+s}$.
   (b) For $i = 1,\ldots,N$, the parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![\alpha_i]\!]_{k+s}$.
   (c) *Compute linear combinations:*
      i. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $([\![\alpha_1]\!]_{k+s},\ldots,[\![\alpha_N]\!]_{k+s}, [\![\beta_1]\!]_{k+s},\ldots,[\![\beta_M]\!]_{k+s})$ and $([\![r \cdot z_1]\!]_{k+s},\ldots,[\![r, \cdot z_N]\!]_{k+s}, [\![r \cdot v_1]\!]_{k+s},\ldots,[\![r \cdot v_M]\!]_{k+s})$ to obtain $[\![u]\!]_{k+s} = [\![\sum_{i=1}^N \alpha_i \cdot (r \cdot z_i) + \sum_{m=1}^M \beta_m \cdot (r \cdot v_m)]\!]_{k+s}$.
      ii. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $(\alpha_1,\ldots,\alpha_N,\beta_1,\ldots,\beta_M)$ and $([\![z_1]\!]_{k+s},\ldots,[\![z_N]\!]_{k+s}, [\![v_1]\!]_{k+s},\ldots,[\![v_M]\!]_{k+s})$ to obtain $[\![w]\!]_{k+s} = [\![\sum_{i=1}^N \alpha_i \cdot z_i + \sum_{m=1}^M \beta_m \cdot v_m]\!]_{k+s}$.
   (d) The parties run $\mathsf{open}([\![r]\!]_{k+s})$ to receive r.
   (e) Each party locally computes $[\![T]\!]_{k+s} = [\![u]\!]_{k+s} - r \cdot [\![w]\!]_{k+s}$.
   (f) The parties call $\mathcal{F}_{\text{CheckZero}}(k+s)$ on $[\![T]\!]_{k+s}$. If $\mathcal{F}_{\text{CheckZero}}(k+s)$ outputs reject, the parties output $\perp$ and abort. If it outputs accept, they proceed.
6. *Output reconstruction:* For each output wire of the circuit with $[\![v]\!]_{k+s}$, the parties locally convert to $[\![v]\!]_k$. Then, they run $v \bmod 2^k = \mathsf{open}([\![v]\!]_k, j)$, where P_j is the party whose output is on the wire. If P_j received $\perp$ from the open procedure, then it sends $\perp$ to the other parties, outputs $\perp$ and halts.

Output: If a party has not aborted, then it outputs the values received on its output wires.

Unfortunately, the fact that the underlying multiplication protocol is secure only up to additive attacks means that the output of the multiplications might be incorrect. Thus, before reconstructing the outputs, the parties run a short verification step which guarantees that if cheating took place, the honest parties will detect it and abort. This is achieved by having the parties first taking a random linear combination of the shares on output wires of all multiplication gates and the shares on input wires, i.e. computing $[\![u]\!] = [\![\sum_{i=1}^{N} \alpha_i \cdot z_i + \sum_{m=1}^{M} \beta_m \cdot v_m]\!]$, and taking a random linear combination of the *randomized* sharing on these wires, i.e., computing $[\![w]\!] = [\![\sum_{i=1}^{N} \alpha_i \cdot (r \cdot z_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m)]\!]$, where N is the number of multiplication gates, M is the number of input wires and all α_i and β_m are random secrets. Then, the parties check that $[\![T]\!] = [\![w]\!] - r \cdot [\![u]\!]$ is a sharing of 0 using the ideal functionality $\mathcal{F}_{\text{CheckZero}}$.

The protocol as described so far works directly for circuits which are defined over a finite field $\mathbb{F}$. As shown in [CGH+18], if the adversary carries out an additive attack in any of the multiplication, the check will pass for exactly one choice of r or a random coefficient, resulting with a cheating probability of $3/|\mathbb{F}|$. However, this does not work when moving to rings. To see this, assume that the adversary has attacked exactly one gate, indexed by i_0, such that $z_{i_0} = x_{i_0} \cdot y_{i_0} + d_{i_0}$ and $r \cdot z_{i_0} = (r \cdot x_{i_0}) \cdot y_{i_0}$ (i.e., the adversary added d_i to the result of multiplying $x_{i_0} \cdot y_{i_0}$ and acted honestly when multiplying $r\dot{x}_{i_0}$ with y_{i_0}). For simplicity assume that the output of this gate is an output wire of the circuit. Thus, we have that $T = (r \cdot x_{i_0}) \cdot y_{i_0} - r \cdot (x_{i_0} \cdot y_{i_0} + d_{i_0}) = r \cdot d_{i_0}$. Now, when working over fields, $T = 0$ only if $r = 0$ (since $d_{i_0} \neq 0$), which happens with probability $1/|\mathbb{F}|$. However, when working over the ring $\mathbb{Z}_{2^k}$, the adversary can choose $d_{i_0} = 2^{k-1}$, which means that $T \equiv_k 0$ if r is even, which happens with probability $1/2$.

In order to reduce the cheating success probability, we borrow the idea of [CDE+18] to work on the larger ring $\mathbb{Z}_{2^{k+s}}$. This solves the above attack which now can succeed with probability $1/2^{s+1}$ only (since now $r \cdot d_{i_0} \equiv_{k+s} 0$ for $d_{i_0} = 2^{k-1}$ is equivalent to $r \equiv_{k+s-(k-1)} 0$, i.e., for this to hold the adversary needs to guess the upper $s+1$ bits of r). More generally, we show in Lemma 4.2 that for any attack in any of the calls to $\mathcal{F}_{\mathsf{mult}}$ with an additive value $d \not\equiv_k 0$, the honest parties will output accept at the end of the verification step with probability of at most $2^{-s+\log(s+1)}$. On the other hand, the adversary may now also carry out attacks with additive values that are congruent to 0 modulo 2^k but not modulo 2^{k+s}. While this has no effect on the correctness of the output (since it does not change the lower k bits of the values on the wires), a challenge here is to show that it is possible to simulate correctly when $\mathcal{F}_{\text{CheckZero}}$ returns accept or reject. In Theorem 4.3, where we prove the security of our compiler, we show that there are several cases here and that the simulation has the same distribution as in the real execution.

Finally, we want to highlight another subtle issue regarding the security of the protocol. As can be seen in the description of the protocol, for the random linear combination taken in the verification step, we require the random coefficients to remain secret during the computation (thus producing them using the functionality $\mathcal{F}_{\mathsf{rand}}$). We stress that this is essential for keeping the

protocol secure. In particular, if the coefficients were revealed to the parties, then the adversary will be able to carry out a selective failure attack where one bit of information is revealed by $\mathcal{F}_{\text{CheckZero}}$. To see this, assume again that the adversary has attacked exactly one gate, indexed by i_0, in the following way. When multiplying x_{i_0} with y_{i_0}, the adversary acted honestly, but when multiplying $r \cdot x_{i_0}$ with y_{i_0}, it added the value d_{i_0}. Thus, on the output wire, the parties hold a sharing of the pair $(x_{i_0} \cdot y_{i_0}, r \cdot x_{i_0} \cdot y_{i_0} + d_{i_0})$. Now, assume that this wire enters another multiplication gate, indexed by j_0 with input shares on the second wire being $(w_{j_0}, r \cdot w_{j_0})$ and that the output of this second gate is an output wire of the circuit. Thus, on the output of this gate, the parties will hold the sharing $(x_{i_0} \cdot y_{i_0} \cdot w_{j_0}, (r \cdot x_{i_0} \cdot y_{i_0} + d_{i_0})w_{j_0})$ (assuming the adversary does not attack this gate as well). In this case, we have that $T = \alpha_{i_0} \cdot d_{i_0} + \alpha_{j_0} \cdot (d_{i_0} \cdot w_{j_0}) = d_{i_0}(\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0})$. Now, if $d_{i_0} = 2^{k+s-1}$ then it follows that $T \equiv_{k+s} 0$ if and only if $\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0}$ is even.

The attack presented above does not change the k lower bits of the values on the wires, and thus has no effect on the correctness of the output. However, if α_{i_0} and α_{j_0} are public and known to the adversary, then by $\mathcal{F}_{\text{CheckZero}}$'s ouptut the adversary may be able to learn whether w_{j_0} is even or not. In contrast, when α_{i_0} and α_{j_0} are kept secret, learning whether $\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0}$ is even or odd does not reveal any information about w_{j_0} since it is now perfectly masked by α_{i_0} and α_{j_0}. Therefore, to prevent this type of attack, we are forced to use random secrets for our random linear combination. Here is where the functionality $\mathcal{F}_{\mathsf{DotProduct}}$ becomes handy, as it allows to compute the sum of products of sharings in an efficient way which is exactly what we need to compute $\sum_{i=1}^{N} [\![\alpha_i]\!] \cdot [\![z_i]\!]$.

Lemma 4.2. *If $\mathcal{A}$ sends an additive value $d \not\equiv_k 0$ in any of the calls to $\mathcal{F}_{\mathsf{mult}}$ in the execution of Protocol 4.1, then the value T computed in the verification stage of Step 5 equals 0 with probability $2^{-s+\log(s+1)}$.*

Proof: Suppose that $([\![x_i]\!]_{k+s}, [\![y_i]\!]_{k+s}, [\![z_i]\!]_{k+s})$ is the multiplication triple corresponding to the i-th multiplication gate, where $[\![x_i]\!]_{k+s}, [\![y_i]\!]_{k+s}$ are the sharings on the input wires and $[\![z_i]\!]_{k+s}$ is the sharing on the output wire. We note that the values on the input wires may not actually be the appropriate values as when the circuit is computed by honest parties. However, in the verification step, each gate is examined separately, and all that is important is whether the randomized result is $[\![r \cdot z_i]\!]_{k+s}$ for whatever z_i is here (i.e., even if an error was added by the adversary in previous gates). By the definition of $\mathcal{F}_{\mathsf{mult}}$, a malicious adversary is able to carry out an additive attack, meaning that it can add a value to the output of each multiplication gate. We denote by $\delta_i \in \mathbb{Z}_{2^{k+s}}$ the value that is added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with $[\![x_i]\!]_{k+s}$ and $[\![y_i]\!]_{k+s}$, and by $\gamma_i \in \mathbb{Z}_{2^{k+s}}$ the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with the shares $[\![y_i]\!]_{k+s}$ and $[\![r \cdot x_i]\!]_{k+s}$. However, it is possible that the adversary has attacked previous gates and so $[\![y_i]\!]_{k+s}$ is actually multiplied with $[\![r \cdot x_i + \epsilon_i]\!]$, where the value $\epsilon_i \in \mathbb{Z}_{2^{k+s}}$ is an accumulated error from previous gates [9]. Thus, it holds

[9] Although attacks in previous gates may be carried out on both multiplications, the idea is here is to fix x_i which is shared by $[\![x_i]\!]_{k+s}$ at the current value on the wire, and

that $\mathsf{val}(\llbracket z_i \rrbracket)^H = x_i \cdot y_i + \delta_i$ and $\mathsf{val}(\llbracket r \cdot z_i \rrbracket)^H = (r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i$. Similarly, for each input wire with sharing $\llbracket v_m \rrbracket$, it holds that $\mathsf{val}(\llbracket r \cdot v_m \rrbracket)^H = r \cdot v_m + \xi_m$, where $\xi_m \in \mathbb{Z}_{2^{k+s}}$ is the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with $\llbracket r \rrbracket_{k+s}$ and the shared input $\llbracket v_m \rrbracket_{k+s}$. Thus, we have that

$$\mathsf{val}(\llbracket u \rrbracket)^H = \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \Theta_1$$
$$\mathsf{val}(\llbracket w \rrbracket)^H = \sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2$$

where $\Theta_1 \in \mathbb{Z}_{2^{k+s}}$ and $\Theta_2 \in \mathbb{Z}_{2^{k+s}}$ are the values being added by the adversary when $\mathcal{F}_{\mathsf{DotProduct}}$ is called in the verification step, and so

$$\begin{aligned}
\mathsf{val}(\llbracket T \rrbracket)^H &= \mathsf{val}(\llbracket u \rrbracket)^H - r \cdot \mathsf{val}(\llbracket w \rrbracket)^H = \\
&= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \theta_1 \\
&\qquad - r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2 \right) \\
&= \sum_{i=1}^{N} \alpha_i \cdot (\epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i) + \sum_{m=1}^{M} \beta_m \cdot \xi_m + (\Theta_1 - r \cdot \Theta_2), \quad (1)
\end{aligned}$$

where the second equality holds because r is opened and so the multiplication $r \cdot \llbracket w \rrbracket_{k+s}$ always yields $\llbracket r \cdot w \rrbracket_{k+s}$. Let $\Delta_i = \epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i$.

Our goal is to show that $\mathsf{val}(\llbracket T \rrbracket)^H$, as shown in Eq. (1), equals 0 with probability at most $2^{-s+\log(s+1)}$. We have the following cases.

– *Case 1: There exists $m \in [M]$ such that $\xi_m \not\equiv_k 0$.* Let m_0 be the smallest such m for which this holds. Then $\mathsf{val}(\llbracket T \rrbracket)^H \equiv_{k+s} 0$ if and only if

$$\beta_{m_0} \cdot \xi_{m_0} \equiv_{k+s} \left(- \sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).$$

Let 2^u be the largest power of 2 dividing ξ_{m_0}. Then we have that

$$\beta_{m_0} \equiv_{k+s-u} \left(\frac{- \sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2)}{2^u} \right) \cdot \left(\frac{\xi_{m_0}}{2^u} \right)^{-1}.$$

By the assumption that $\xi_m \not\equiv_k 0$ it follows that $u < k$ and so $k + s - u > s$ which means that the above holds with probability at most 2^{-s}, since β_{m_0} is uniformly distributed over $\mathbb{Z}_{2^{k+s}}$.

---

then given the randomized sharing $\llbracket x'_i \rrbracket_{k+s}$, define $\epsilon_i = x'_i - r \cdot x_i$ as the accumulated error on the input wire.

– *Case 2: All* $\xi_m \equiv_k 0$. By the assumption in the lemma, some additive value $d \not\equiv_k 0$ was sent to $\mathcal{F}_{\mathsf{mult}}$. Since none was sent for the input randomization, there exists some $i \in \{1, \ldots, N\}$ such that $\delta_i \not\equiv_k 0$ or $\gamma_i \not\equiv_k 0$. Let i_0 be the smallest such i for which this holds. Note that since this is the first error added which is $\not\equiv_k 0$, it holds that $\epsilon_{i_0} \equiv_k 0$. Thus, in this case, $\mathsf{val}(\llbracket T \rrbracket)^H \equiv_{k+s} 0$ if and only if $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+s} Y$, where

$$Y = \left(-\sum_{\substack{i=1 \\ i \neq i_0}}^{N} \alpha_i \cdot \Delta_i - \sum_{m=1}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).$$

Let q be the random variable corresponding to the largest power of 2 dividing Δ_{i_0}, where we define $q = k + s$ in the case that $\Delta_{i_0} \equiv_{k+s} 0$. Let E denote the event $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+s} Y$. We have the following claims.

- *Claim 1: For* $k < j \leq k + s$*, it holds that* $\Pr[q = j] \leq 2^{-(j-k)}$.
  To see this, suppose that $q = j$ and $j > k$. It holds then that $\Delta_{i_0} \equiv_j 0$, and so $\Delta_{i_0} \equiv_k 0$. We first claim that in this case it must hold that $\delta_{i_0} \not\equiv_k 0$. Assume in contradiction that $\delta_{i_0} \equiv_k 0$. In addition, by our assumption we have that $\gamma_{i_0} \not\equiv_k 0$, $\epsilon_i \equiv_k 0$ and $\Delta_{i_0} = \epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0} - r \cdot \delta_{i_0} \equiv_k 0$. However, $\epsilon_i \cdot y_{i_0} \equiv_k 0$ and $r \cdot \delta_{i_0} \equiv_k 0$ imply that $\gamma_{i_0} \equiv_k 0$, which is a contradiction. We thus assume that $\delta_{i_0} \not\equiv_k 0$, and in particular there exists $u < k$, such that u is the largest power of 2 dividing δ_{i_0}. It is easy to see then that $q = j$ implies that $r \equiv_{j-u} \left(\frac{\epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0}}{2^u} \right) \cdot \left(\frac{\delta_{i_0}}{2^u} \right)^{-1}$. Since $r \in \mathbb{Z}_{2^{k+s}}$ is uniformly random and $u < k$, we have that this equation holds with probability of at most $2^{-(j-u)} \leq 2^{-(j-k)}$.
- *Claim 2: For* $k < j < k + s$ *it holds that* $\Pr[E \mid q = j] \leq 2^{-(k+s-j)}$.
  To prove this let us assume that $q = j$ and that E holds. In this case we can write $\alpha_{i_0} \equiv_{k+s-j} \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j} \right)^{-1}$. For $k < j < k + s$ it holds that $0 < k + s - j < s$ and therefore this equation can be only satisfied with probability at most $2^{-(k+s-j)}$, given that $\alpha_{i_0} \in \mathbb{Z}_{2^s}$ is uniformly random.
- *Claim 3:* $\Pr[E \mid 0 \leq q \leq k] \leq 2^{-s}$.
  This is implied by the proof of the previous claim, since in the case that $q = j$ with $0 \leq j \leq k$, it holds that $k + s - j \geq s$, so the event E implies that $\alpha_{i_0} \equiv_s \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j} \right)^{-1}$, which holds with probability at most 2^{-s}.

Putting these pieces together, we thus have the following:

$$\begin{aligned} \Pr[E] &= \Pr[E \mid 0 \leq q \leq k] \cdot \Pr[0 \leq q \leq k] + \sum_{j=k+1}^{k+s} \Pr[E \mid q = j] \cdot \Pr[q = j] \\ &\leq 2^{-s} + s \cdot 2^{-s} = (s+1) \cdot 2^{-s} = 2^{-s+\log(s+1)}. \end{aligned} \tag{2}$$

To sum up the proof, in the first case we obtained that $T = 0$ with probability of at most 2^{-s} whereas in the second case, this holds with probability of at most $2^{-s+\log(s+1)}$. Therefore, we conclude that the probability that $T = 0$ in

the verification step is bounded by $2^{-s+\log(s+1)}$ as stated in the lemma. This concludes the proof. ■

We are now ready to prove the security of Protocol 4.1. The proof appears in Section B.

Theorem 4.3. *Let f be an n-party functionality over $\mathbb{Z}_{2^k}$ and let s be a statistical security parameter. Then, Protocol 4.1 securely computes f with abort in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\text{CheckZero}})$-hybrid model with statistical error $2^{-s+\log(s+1)}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Concrete efficiency. We now analyze the performance of the protocol. Recall that M is the number of inputs and N is the number of multiplication gates in the circuit. We denote by O the number of output wires of the circuit, and for a given functionality $\mathcal{F}_*(\ell)$, we denote by $\mathcal{C}_*(\ell)$ the communication cost (in bits) of calling this primitive.

For each input wire, we have one call to $\mathcal{F}_{\text{input}}(k+s)$, which is translated into one call to $\mathcal{F}_{\mathsf{rand}}(k+s)$, one call to $\mathsf{open}(\llbracket r \rrbracket_{k+s}, i)$ and one element in $\mathbb{Z}_{2^{k+s}}$ that is sent by some party P_i to the other parties. In addition, there is one call to $\mathcal{F}_{\mathsf{mult}}(k+s)$ to randomize each input. This adds up to $M \cdot (2 \cdot \mathcal{C}_{\mathsf{rand}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + (k+s))$.

For each multiplication gate, we call $\mathcal{F}_{\mathsf{mult}}(k+s)$ twice. Then, in the verification step, $\mathcal{F}_{\mathsf{rand}}(k+s)$ is called for each input wire and multiplication gate. This adds $N \cdot (\mathcal{C}_{\mathsf{rand}} + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s))$. The remaining of the verification step consists of two calls to $\mathcal{F}_{\mathsf{DotProduct}}(k+s)$, one call to $\mathsf{open}(\llbracket r \rrbracket_{k+s})$ and one call to $\mathcal{F}_{\text{CheckZero}}(k+s)$. Recall that we assume that the protocol realizing $\mathcal{F}_{\mathsf{DotProduct}}(k+s)$ has the same communication complexity as $\mathcal{F}_{\mathsf{mult}}(k+s)$, so this adds up to $2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + \mathcal{C}_{\mathsf{CheckZero}}(k+s)$. However, as these are small constants which do not depend on the size of the circuit, we exclude them from the final count. In the output reconstruction step, for each output wire, there is one call to $\mathsf{open}(\llbracket v \rrbracket_k, i)$.

We thus have that the cost of the protocol is

$$\begin{aligned} M \cdot \big(2 \cdot \mathcal{C}_{\mathsf{rand}}(k+s) + \mathcal{C}_{\mathsf{mult}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + (k+s)\big) \\ + N \cdot (\mathcal{C}_{\mathsf{rand}}(k+s) + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s)) + O \cdot \mathcal{C}_{\mathsf{open(i)}}(k). \end{aligned}$$

For circuits where $N \gg M, O$ (i.e., there are much more multiplication gates than input and output wires), this is translated to $N \cdot (\mathcal{C}_{\mathsf{rand}}(k+s) + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s))$. Notice that for some instantiations, like the replicated secret sharing based one from Section 5, $\mathcal{F}_{\mathsf{rand}}$ is "free" in the sense that it can be implemented efficiently by relying on a computational assumption, e.g., PRGs with correlated keys.

Basic Primitives for Secure Computation. We conclude this section with a short discussion about primitives for secure computation like comparison and truncation, among others, which are of importance in many applications of secure computation like private machine learning or flow-control in MPC programs.

The study of basic primitives for MPC has a rich history, including some works as [Cd10,MR18,DEF+19,DFK+06]. However, most of these works are concerned with the case of MPC over fields and as such they face different challenges and provide different solutions. For example, a very simple operation that arises in these primitives over fields is dividing by powers of 2, which is achieved over fields of odd characteristic by simply multiplying (locally) by the inverse of this number. However, over $\mathbb{Z}_{2^k}$ this is not so straightforward, which complicates the extension of these techniques to the ring case.

Recent work has studied the development of basic primitives over rings [MR18,DEF+19]. In particular, the work of [DEF+19] has shown that, in spite of being more general than fields (and hence more complex), rings offer several benefits for many of the basic primitives considered in the literature. Intuitively, this stems from the fact that $\mathbb{Z}_{2^\ell}$ is inherently more "compatible" with bits, which is what these primitives are mostly concerned with. Hence, it is natural to analyze whether or not our compiler supports these basic primitives.

We first observe that our check-to-zero protocol from Section 3.3 is already an instantiation of a basic primitive. Furthemore, just like we adapted this check-to-zero from [DEF+19] to our setting, other techniques from that work can be easily incorporated into ours in order to provide bit-decomposition and bit-extraction, truncations and signed comparisons. At the heart of these primitives lies the generation of random shared bits, which as we saw in Section 3.3, extends smoothly to the setting of an arbitrary secret sharing scheme. The fact that shares can be converted from mod 2^k to mod 2 also plays an important role, and a converse conversion can be envisioned using the ideas from [DEF+19].

However, we stress that all of this comes at the expense of using the expensive $\mathcal{F}_{\text{CorrectMult}}$ for all of the multiplication calls. A natural question to ask is whether it is possible to use $\mathcal{F}_{\mathsf{mult}}$, which can be realized very efficiently, instead of $\mathcal{F}_{\text{CorrectMult}}$. Answering this question is beyond the scope of this work and is left as an open problem.

5 Replicated-SS-based Instantiation for Three Parties

We now present briefly an efficient three party instantiation of our compiler from replicated secret sharing. Sharing a value $x \in \mathbb{Z}_{2^\ell}$ is done by picking at random $x_1, x_2, x_3 \in \mathbb{Z}_{2^\ell}$ such that $\sum_i x_i \equiv_\ell x$. P_i's share of x is the pair (x_i, x_{i+1}) and we use the convention that $i + 1 = 1$ when $i = 3$. To reconstruct a secret, P_i receives the missing share from the two other parties. Note that reconstructing a secret is robust in the sense that parties either reconstruct the correct value x or they abort.

Replicated secret sharing satisfies the properties described in Section 2.1, and one can efficiently realize the required functionalities described in the same section. Specifically, as shown in [MR18,AFL+16,LN17], $\mathcal{F}_{\mathsf{rand}}$ can be realized with out any communication and $\mathcal{F}_{\mathsf{mult}}$ can be realized by having each party sending one ring element. Further more, in [CGH+18], it was shown that $\mathcal{F}_{\mathsf{DotProduct}}$ can be computed at the same cost of $\mathcal{F}_{\mathsf{mult}}$. In addition, $\mathcal{F}_{\text{CheckZero}}$ can be realized very

efficiently by relying on a Random Oracle $\mathcal{H}$. The observation we rely on is that, if $\sum_i x_i \equiv_\ell 0$, then $x_{i-1} \equiv_\ell -(x_i + x_{i+1})$ and so P_i can send $z_i = \mathcal{H}(-(x_i + x_{i+1}))$ which will be equal to x_{i-1} that is held by P_{i+1} and P_{i-1}. Since only one party is corrupted, it suffices that each P_i will send it only to P_{i+1}. Upon receiving z_i from P_i, P_{i+1} checks that $z_i = \mathcal{H}(x_{i-1})$ and aborts if this is not the case. For completeness, we present all the protocols in Section C.

Efficiency Analysis. Using the analysis from Section 4, we know that the amortized communication complexity per multiplication gate is $\mathcal{C}_{\mathsf{rand}}(k+s) + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s)$. In our case $\mathcal{C}_{\mathsf{rand}}(k+s) = 0$, and $\mathcal{C}_{\mathsf{mult}}(k+s) = 3 \cdot (k+s)$, so the overall amortized communication per multiplication is of only $6 \cdot (k+s)$ bits. For each party this translates to sending $2(k+s)$ bits for each multiplication.

6 Shamir-SS-based Instantiation for Any Number of Parties

In this section, we present our instantiation based on Shamir's secret sharing over rings, using the techniques from [ACD+19]. Over finite fields, Shamir's scheme requires a distinct evaluation point for each player, and one more for the secret. This is usually not a problem if the size of the field is not too small. However, over commutative rings R the condition on the sequence of evaluation points $\alpha_0, \ldots, \alpha_n \in R$ is that the pairwise difference $\alpha_i - \alpha_j$ is invertible for each pair of indices $i \neq j$. For our ring of interest $\mathbb{Z}_{2^\ell}$, the largest such sequence the ring admits is only of length 2 (e.g. $(\alpha_0, \alpha_1) = (0, 1)$).

The solution from [ACD+19] is to embed inputs from $\mathbb{Z}_{2^\ell}$ into a large enough Galois ring R that has $\mathbb{Z}_{2^\ell}$ as a subring. This ring is of the form $R = \mathbb{Z}_{2^\ell}[X]/(h(X))$, where $h(X)$ is a monic polynomial of degree $d = \lceil \log_2 n \rceil$ such that $h(X) \bmod 2 \in \mathbb{F}_2[X]$ is irreducible. Elements of R thus correspond uniquely to polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ that are of degree at most $d-1$. Note the similarity between the Galois ring and finite field extensions of $\mathbb{F}_2$: elements of the finite field $\mathbb{F}_{2^d}$ correspond uniquely to polynomials of at most degree $d-1$ with coefficients in $\mathbb{F}_2$.

There is a ring homomorphism $\pi : R \to \mathbb{Z}_{2^\ell}$ that sends $a_0 + a_1 X + \cdots + a_{d-1} X^{d-1} \in R$ to the free coefficient a_0, which we shall use later on.[10] For more relevant structural properties of Galois rings, see [ACD+19].

We adopt the above-mentioned version of Shamir's scheme over R, but restrict the secret space to $\mathbb{Z}_{2^\ell}$. The share space will be equal to R. Let $1 \leq \tau \leq n$ be the privacy parameter of the scheme. Then, the set of *correct* share vectors is

$$C_\tau = \{(f(\alpha_1), \ldots, f(\alpha_n)) \in R^n \mid f \in R[X],\ \deg(f) \leq \tau,\ \text{and } f(\alpha_0) \in \mathbb{Z}_{2^\ell} \subset R\}\,. \tag{3}$$

With the restriction that the secret is in $\mathbb{Z}_{2^\ell}$, we have that C_τ is an $\mathbb{Z}_{2^\ell}$-module, i.e. the secret-sharing scheme is $\mathbb{Z}_{2^\ell}$-linear. Since it is based on polynomial

[10] Technically, an element of R is a residue class modulo the ideal $(h(X))$, but we omit this for simplicity of notation.

interpolation, the properties from 2.1 can be easily seen to hold. This includes division by 2 if all the shares are even.

In this section, we denote a sharing under C_τ as $[\![x]\!] = (x_1, \ldots, x_n)$. We call τ the *degree* of the sharing. The reason we are explicit about τ is that we will use sharings of two different degrees. This stems from the critical property of this secret-sharing scheme that enables us to evaluate arithmetic circuits: this secret-sharing scheme is *multiplicative*. This means there is a $\mathbb{Z}_{2^\ell}$-linear map $R^n \to \mathbb{Z}_{2^\ell}$ that for sharings $[\![x]\!], [\![y]\!]$ sends $(x_1y_1, \ldots, x_ny_n) \mapsto x \cdot y$.

Put differently, $(x_1y_1, \ldots, x_ny_n) \in C_{2\tau}$ is a degree-2τ sharing with secret $x \cdot y$. We denote it $[\![x \cdot y]\!]_{(2\tau)} = (x_1y_1, \ldots, x_ny_n)$ — in particular note the parenthesized subscript refers to the degree of the sharing, as opposed to the modulus. Note that $C_i \subseteq C_j$ for $0 < i < j$; in particular every degree-2τ sharing is also a sharing of degree $n-1$. A sharing of degree $n-1$ is related to additive secret sharing, where the secret equals the sum of the shares $x = \sum_i x_i$. The difference is that here there are constants, i.e. we may write $x = \sum_i \lambda_i x_i$, for $\lambda_1, \ldots, \lambda_n \in R$. We shall make use of this in our multiplication protocol, ensuring that parties only need to communicate an element of $\mathbb{Z}_{2^\ell}$ instead of an element of R. However, note that $[\![\cdot]\!]_{(2\tau)}$ does not meet the definition of a secret-sharing scheme in Section 2.1, in particular because the corrupted parties shares are not well defined and cannot be computed from the honest parties' shares.

6.1 Generating Randomness

We efficiently realize $\mathcal{F}_{\mathsf{rand}}$ by letting each player P_i sample and secret-share a random element s_i, and then multiplying the resulting vector of n random elements with a particular[11] Vandermonde matrix [DN07].[12] Of the resulting vector, τ entries are discarded to ensure the adversary has zero information about the remaining ones. Thus, $n - \tau$ random elements are outputted, resulting in an amortized communication cost of $O(n)$ ring elements per element. A priori the adversary can cause the sharings to be incorrect; this is remedied with Protocol 6.3 by opening a random linear combination of the sharings and verifying the result.

Since our secret-sharing scheme $[\![\cdot]\!]$ is $\mathbb{Z}_{2^\ell}$-linear, we would like to choose our matrix with entries in $\mathbb{Z}_{2^\ell}$. Unfortunately, the Vandermonde matrix we need does not exist over $\mathbb{Z}_{2^\ell}$, for the same reason secret sharing does not work. However, the secret-sharing scheme which consists of d parallel sharings of $[\![\cdot]\!]$ be interpreted as an R-linear secret-sharing scheme [CCXY18,ACD+19]. This secret-sharing scheme, which we denote as $\langle\cdot\rangle$, has share space S^d (since the scheme is identical to sharing d independent secrets in S in parallel using $[\![\cdot]\!]$), and secret space R^d. The scheme is R-linear because the module of share vectors, which is $(C_\tau)^d$, is an R-module via the tensor product $(C_\tau)^d \cong C_\tau \otimes_S S^d \cong C_\tau \otimes_S R$. In practice, a single secret-shared element $\langle x \rangle$ may be interpreted as a secret-shared

[11] Over fields this can be a general Vandermonde matrix, but this is not sufficient over R.

[12] In general, any R-linear code with good distance and dimension suffices to get $O(n)$ complexity in the protocol, but the Vandermonde construction is optimal.

column vector $(\llbracket x_1 \rrbracket, \ldots, \llbracket x_d \rrbracket)^T$. To compute the action of an element $r \in R$ on $\langle x \rangle$ in this representation, we first need to fix a basis of R over S. Recall $R = \mathbb{Z}_{2^\ell}[X]/(h(X))$, so we may pick the canonical basis $1, X, \ldots, X^{d-1} \in R$. This allows us to represent an element $a \in R$ as a column vector $(a_0, \ldots, a_{d-1})^T \in S^d$, i.e. explicitly: $a = a_0 + a_1 X + \cdots + a_{d-1} X^{d-1}$. Multiplication by $r \in R$ is an S-linear map of vectors $S^d \to S^d$, i.e. it can be represented as a $d \times d$ matrix M_r with entries in S. The product $r \langle x \rangle = \langle rx \rangle$ is then equal to $M_r(\llbracket x_1 \rrbracket, \ldots, \llbracket x_d \rrbracket)^T$. If a single party P has a vector of shares $(s_1, \ldots, s_d) \in R$ for $\langle x \rangle = (\llbracket x_1 \rrbracket, \ldots, \llbracket x_d \rrbracket)^T$, then $M_r(s_1, \ldots, s_d)^T$ is their vector of shares corresponding to $\langle rx \rangle$.

In our protocol, the parties will calculate a matrix-vector product $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T = A(\langle s_1 \rangle, \ldots, \langle s_n \rangle)^T$, where A has entries in R. This can be computed by writing out the R-linear combinations $\langle r_i \rangle = \sum_{k=1}^{n} a_{ik} \langle s_k \rangle = \sum_{k=1}^{n} M_{a_{ik}} \langle s_k \rangle$, with $\langle s_k \rangle = (\llbracket s_{k1} \rrbracket, \llbracket s_{kd} \rrbracket)^\intercal$. Fix a sequence $\beta_1, \ldots, \beta_n \in R$ such that for each pair of indices $i \neq j$ we have that $\beta_i - \beta_j$ is invertible.[13] We let A be the $(n - \tau) \times n$ matrix such that the j-th column is $(1, \beta_j, \beta_j^2, \ldots, \beta_j^{n-\tau-1})^T$. This matrix is hyperinvertible, i.e. any square submatrix is invertible [ACD+19].

PROTOCOL 6.1 (Generating random sharings of $\llbracket \cdot \rrbracket$)

1. Each party P_i samples an element $s_i \leftarrow (\mathbb{Z}_{2^\ell})^d$ and secret-shares it as $\langle s_i \rangle$ among all parties.
2. The parties locally compute the linear matrix-vector product to obtain $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T := A(\langle s_1 \rangle, \ldots, \langle s_n \rangle)^T$.
3. The parties execute Protocol 6.3 $\lceil \kappa / d \rceil$ times in parallel on $\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle$If any execution fails, they abort. Otherwise, for each $j = 1, \ldots, n - \tau$ they interpret $\langle r_j \rangle = (\llbracket r_{j1} \rrbracket, \ldots, \llbracket r_{jd} \rrbracket)$ and output $\llbracket r_{11} \rrbracket, \ldots, \llbracket r_{1d} \rrbracket, \llbracket r_{21} \rrbracket, \ldots, \llbracket r_{(n-\tau)d} \rrbracket$.

Lemma 6.2. *Protocol 6.1 securely computes $(n - \tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{rand}}$ for $\llbracket \cdot \rrbracket$ with statistical error of at most $2^{-\kappa}$ in the presence of a malicious adversary controlling $t < n/2$ parties.*

The proof is in Section D.1

6.2 Checking Correctness of Sharings

We check whether sharings are correct by taking a random linear combination of the sharings, masking it with a random sharing, and opening the result to all parties.

This protocol does not securely compute an ideal functionality, because privacy is not preserved if the sharings are incorrect. The way we use it this does not matter, since we only verify correctness of sharings of random elements.

[13] We may just use $(\beta_1, \ldots, \beta_n) = (\alpha_1, \ldots, \alpha_n)$.

PROTOCOL 6.3 (Checking correctness of sharings of $\langle\cdot\rangle$)

- **Input:** possibly incorrect sharings $\langle x_1\rangle, \ldots, \langle x_N\rangle$, and a possibly incorrect sharing $\langle r\rangle \leftarrow (\mathbb{Z}_{2^\ell})^d$ of a random element
- **Protocol:**
  1. The parties call $\mathcal{F}_{\mathsf{coin}}$ N times to get $a_1, \ldots, a_N \leftarrow (\mathbb{Z}_{2^\ell})^d$.
  2. The parties compute $\langle u\rangle := a_1 \langle x_1\rangle + \cdots + a_N \langle x_N\rangle + \langle r\rangle$.
  3. The parties run $\mathsf{open}(\langle u\rangle)$. If it returns $\perp$, output $\perp$. Else, output $\mathsf{correct}$.

Lemma 6.4. *If at least one of the input sharings $\langle x_1\rangle, \ldots, \langle x_N\rangle$ is incorrect, Protocol 6.3 outputs* $\mathsf{correct}$ *with probability at most $\frac{1}{2^d}$.*

To show correctness, we use the following consequence from [ACD+19, Lemma 3].

Lemma 6.5. *Let $C \subseteq R^n$ be a free R-module. Then for all $x \notin C$ and $u \in R^n$, we have that*

$$\Pr_{r \leftarrow R}[rx + u \in C] \leq \frac{1}{2^d}$$

where r is chosen uniformly at random from R.

Proof: [Proof of Lemma 6.4] Let C denote the R-module of correct share vectors (such as in Equation (3)). One of the input sharings is incorrect; without loss of generality assume it is $\langle x_1\rangle$. The protocol $\mathsf{open}(\langle u\rangle)$ returns a value not equal to $\perp$ if and only if $\langle u\rangle = a_1 \langle x_1\rangle + (a_2 \langle x_2\rangle + \cdots + a_n \langle x_n\rangle + \langle r\rangle)$ is in C. By Lemma 6.5 this probability is bounded by $1/2$, since a_1 was chosen uniformly at random. Since $\langle u\rangle$ is masked with $\langle r\rangle$, the protocol is private. ∎

6.3 Secure Multiplication up to Additive Attacks

Multiplication follows the outline of the passively secure protocol of [DN07]. The protocol begins with an offline phase, where *random double sharings* are produced, i.e. a pair of sharings $(\llbracket r\rrbracket, \llbracket r\rrbracket_{(2\tau)})$ of the same uniformly random element r shared using polynomials of degree τ and degree 2τ, respectively.

We denote a double sharing as $\llbracket r\rrbracket_{(\tau,2\tau)} := ((r_1, r_1'), \ldots, (r_n, r_n'))$. It is a $\mathbb{Z}_{2^\ell}$-linear secret-sharing scheme with secret space $\mathbb{Z}_{2^\ell}$ and share space $R \oplus R$. The set of correct share vectors is the $\mathbb{Z}_{2^\ell}$-module

$$\{((f(\alpha_1), g(\alpha_1)), \ldots, (f(\alpha_n), g(\alpha_n))) \mid f, g \in R[X],\ f(\alpha_0) = g(\alpha_0) \in \mathbb{Z}_{2^\ell},\ \deg(f) \leq \tau,\ \deg(g) \leq 2\tau\}.$$

Secret-sharing an element r under $\llbracket\cdot\rrbracket_{(\tau,2\tau)}$ involves selecting two uniformly random polynomials of degrees at most τ and 2τ respectively.

To generate sharings in $\llbracket\cdot\rrbracket_{(\tau,2\tau)}$, we essentially use Protocol 6.1. However, this protocol does not securely realize $\mathcal{F}_{\mathsf{rand}}$, since in Lemma 6.2 we use the fact

that the simulator can compute the corrupted parties' shares from the honest parties' shares, which is not the case for the degree-2τ part (hence why $[\![\cdot]\!]_{(2\tau)}$, therefore also $[\![\cdot]\!]_{(\tau,2\tau)}$, does not meet the definition of a secret-sharing scheme in Section 2.1). This will only lead to an additive attack in the online phase, which is why we can still use the protocol.

PROTOCOL 6.6 (**Secure multiplication up to an additive attack**)

- **Inputs:** Parties hold correct sharings $[\![x]\!]$, $[\![y]\!]$
- **Offline phase:** The parties execute Protocol 6.1 for $[\![\cdot]\!]_{(\tau,2\tau)}$ instead of $[\![\cdot]\!]$. They only check correctness for the $[\![\cdot]\!]$ part, and not for the $[\![\cdot]\!]_{(2\tau)}$ part. They obtain a random double sharing $([\![r]\!], [\![r]\!]_{(2\tau)})$.
- **Online phase:**
  1. The parties locally calculate $[\![\delta]\!]_{(2\tau)} := [\![x]\!] \cdot [\![y]\!] - [\![r]\!]_{(2\tau)}$.
  2. Each P_i for $i = 1, \dots 2\tau+1$ sends $u_i := \pi(\lambda_i \delta_i)$ to P_1 (recall $\pi(a_0 + a_1 X + \cdots + a_{d-1} X^{d-1}) = a_0 \in \mathbb{Z}_{2^\ell}$, and the λ_i are constants such that $\sum_{i=1}^{n} \lambda_i \delta_i = \delta$)
  3. P_1 can now reconstruct δ as $\delta = \sum_{i=1}^{n} u_i$.
  4. P_1 broadcasts δ.
  5. The parties locally compute $[\![x \cdot y]\!] = [\![r]\!] + \delta$.

The reason each party sends u_i instead of δ_i to P_1 is two-fold. It saves bandwidth, since only an element of $\mathbb{Z}_{2^\ell}$ needs to be communicated instead of an element of R. More importantly though, if the inputs $[\![x]\!], [\![y]\!]$ are not guaranteed to be correct, then sending full shares δ_i can compromise privacy.

Note that it is important that the random double sharing $[\![r]\!]_{(\tau,2\tau)}$ is guaranteed to be correct.

Lemma 6.7. *Protocol 6.6 securely computes $\mathcal{F}_{\mathsf{mult}}$ with statistical error $\leq 2^{-\kappa}$ in the $\mathcal{F}_{\mathsf{rand}}$-hybrid model in the presence of a malicious adversary controlling $t < n/2$ parties.*

The proof appears in Section D.2.

When evaluating a circuit gate-by-gate using Protocol 6.6, we consider an optimization in which we don't need to execute the broadcast (which might be expensive) for each multiplication, but instead they will perform a broadcast just before opening the values. In the multiplication protocol, P_1 will just send a value (not guaranteed to be the same) to all other parties. Each party P_i keeps track of a hash value h_i of all received values in step 4 of the protocol far. Before opening their outputs, each party P_i sends its hash h_i to all other parties. If any party detects a mismatch, they abort. Note that security up to additive attack is guaranteed only after this procedure succeeds, which is executed before opening the output.

In doing so, we lose the invariant that all secret-shared values are guaranteed to be correct. In other scenarios, as for example the $t < n/3$ setting, this completely

breaks the security of the protocol as shown in [GLS19]. However, this is not a problem in our case since the degree-2τ sharings have no redundancy in them. As shown in [GLS19], this is enough to guarantee the security of the protocol with the deferred check, and the reason is essentially that the shares that the potentially corrupt party P_1 receives are now uniformly random and independent of each other.

Efficiency analysis. For generating randomness, the parties communicate n^2 elements to produce $n - \tau \approx n/2$ sharings. Thus, the amortized communication complexity is $2n$ elements per random sharing generation and 2 per party.

In $\mathcal{F}_{\mathsf{mult}}$, the parties generate two sharings and then in the online phase, the parties communicate again $2n$ elements. Overall, the communication consists of $6n$ elements, which means that each party sends 6 elements.

Since we are working in a larger ring, which blows up each element by a factor of $\log n$, we have that $\mathcal{C}_{\mathsf{rand}}(k+s) = 2(k+s) \cdot n \log n$ and $\mathcal{C}_{\mathsf{mult}}(k+s) = 6(k+s) \cdot n \log n$. Thus, the overall cost per multiplication gate in the circuit is $14(k+s) \cdot n \log n$ bits, while each party sends $14(k+s) \cdot \log n$ bits.

7 Implementation and Experiments

We implemented two instantiations of our compiler: one based on replicated secret sharing for the special case of $n = 3$ and one using the Shamir SS based protocol presented in [ACD+19]. All our experiments where run with two different values for k and s, specifically $k = s = 32$ and $k = s = 64$, as that allows for some natural optimizations in terms of how arithmetic in $\mathbb{Z}_{2^{k+s}}$ is implemented.

In order to demonstrate the practicality of our protocol, we compare both of our implementations against previous work that can be considered state of the art for our setting. In particular, we measure the throughput of the computation phase of our protocols against the highly efficient 3 party protocols presented in [EKO+19] using a similar experimental setup as theirs. See Appendix E for details. Our protocol outperforms the protocols in [EKO+19] in two different WAN settings (continental, where parties are running on machines located on the same continent), and global (where parties are distributed globally). In the LAN setting, we perform as well as [EKO+19], which we attribute to differences in the implementations. In particular, the Sharemind and MP-SPDZ frameworks which are used in [EKO+19] are more mature.

We also run the same benchmarks as in [CGH+18] and compare the results against the field based protocols in that work. This experiment comprises running circuits with a fixed number of multiplications and of varying depths. We found that our replicated instantiation outperform its field based cousin in the LAN setting when the size of modulus (i.e., $k + s$) matches the size of the prime used (which is a 61 bit prime). This is in line with the expectation that arithmetic over $\mathbb{Z}_{2^{k+s}}$ for good choices of $k + s$ is more efficient. For experiments in WAN setting we are performing less well, which we attribute to the fact that elements in our protocol are larger by around $\log n$ bits, and thus, when communication becomes the bottleneck, it is not unexpected that we are less efficient.

Finally, the computation phase of our Shamir SS instantiation performs very well compared to the field version. On the other hand, computing the random double-shares that are needed turns out to be very slow in our protocol compared to [CGH+18]. We found that the main issue here was the cost of local computations and thus expect that it is possible to significantly improve this.

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. pages 843–862, 2017.

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. Theory of Cryptography Conference TCC, 2019.

AFL+16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. pages 805–817, 2016.

BEDK19. Assi Barak, Daniel Escudero, Anders Dalskov, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.

BFO12. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. pages 663–680, 2012.

biu17a. Fast large-scale honest-majority mpc for malicious adversaries, 2017. https://github.com/cryptobiu/MPCHonestMajorityNoTriples.

biu17b. Replicated secret sharing for three parties and arithmetic circuit protocol, 2017. https://github.com/cryptobiu/ReplicatedSecretSharing3PartiesNoTriples.

BKY19. Marina Blanton, Ahreum Kang, and Chen Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. Cryptology ePrint Archive, Report 2019/718, 2019. https://eprint.iacr.org/2019/718.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. pages 192–206, 2008.

BTH06. Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. pages 305–328, 2006.

Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. 13(1):143–202, January 2000.

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.

CCXY18. Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. pages 395–426, 2018.

Cd10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. pages 182–199, 2010.

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. pages 769–798, 2018.

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. pages 34–64, 2018.

CS10. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. pages 35–50, 2010.

DEF+19. I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pages 1325–1343, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

DFK+06. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. pages 285–304, 2006.

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. pages 1–18, 2013.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. pages 572–590, 2007.

DOS18. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. pages 799–829, 2018.

EKO+19. Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. Cryptology ePrint Archive, Report 2019/164, 2019. https://eprint.iacr.org/2019/164.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. pages 225–255, 2017.

GIP+14. Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. pages 495–504, 2014.

GIP15. Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. pages 721–741, 2015.

GL05. Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. 18(3):247–287, July 2005.

GLS19. Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. pages 85–114, 2019.

Gol04. Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

GRW18. S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. pages 59–85, 2018.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. 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. pages 259–276, 2017.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. pages 35–52, 2018.

MZ17. Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. pages 19–38, 2017.

PSL80. Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

WGC18. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: Efficient and private neural network training. Cryptology ePrint Archive, Report 2018/442, 2018. https://eprint.iacr.org/2018/442.

A Proofs for Section 3.3 - Checking Equality to 0

A.1 Proof of Proposition 3.5

Proposition A.1 (Propsition 3.5 - restated). *Protocol 3.3 securely computes functionality $\mathcal{F}_{\text{CorrectMult}}$ with abort and with statistical error 2^{-s} in the $(\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model in the presence of malicious adversaries controlling $t < n/2$ parties.*

Proof: Let $\mathcal{A}$ be the real world adversary who controls a set of corrupted parties $\mathcal{C}$ and let $\mathcal{S}$ be the ideal world simulator. The simulator $\mathcal{S}$ works as follows:

1. Generate the multiplication triple:
   (a) $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{rand}}(\ell+s)$, receiving $[\![a]\!]^{\mathcal{C}}_{\ell+s}, [\![a']\!]^{\mathcal{C}}_{\ell+s}, [\![b]\!]^{\mathcal{C}}_{\ell+s}$ sent by $\mathcal{A}$.
   (b) $\mathcal{S}$ play the role $\mathcal{F}_{\mathsf{mult}}(\ell+s)$, receiving d and d' and the corrupted parties' shares $[\![c]\!]^{\mathcal{C}}_{\ell+s}, [\![c']\!]^{\mathcal{C}}_{\ell+s}$ from $\mathcal{A}$.
   (c) $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{coin}}(s)$ sampling $r \in \mathbb{Z}_{2^s}$ and hands it to $\mathcal{A}$.
   (d) $\mathcal{S}$ computes $[\![a'']\!]^{\mathcal{C}}_{\ell+s} = r \cdot [\![a]\!]^{\mathcal{C}}_{\ell+s} - [\![a']\!]^{\mathcal{C}}_{\ell+s}$, chooses a random $a'' \in \mathbb{Z}_{2^{\ell+s}}$ and chooses random shares for the honest parties, given a'' and $[\![a'']\!]^{\mathcal{C}}_{\ell+s}$. Then, it simulates the honest parties in the execution of $\mathsf{open}([\![a'']\!]_{\ell+s})$. If the honest parties output $\perp$ in the execution, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{CorrectMult}}$ and halts.
   (e) $\mathcal{S}$ computes $[\![w]\!]^{\mathcal{C}}_{\ell+s} = a'' \cdot [\![b]\!]^{\mathcal{C}}_{\ell+s} - r \cdot [\![c]\!]^{\mathcal{C}}_{\ell+s} + [\![c']\!]^{\mathcal{C}}_{\ell+s}$. Then, it sets $w = d' - r \cdot d$ and chooses the honest parties' shares $[\![w]\!]^{H}_{\ell+s}$ accordingly.
   (f) Finally, $\mathcal{S}$ simulates the honest parties in the execution of $\mathsf{open}([\![w]\!]_{\ell+s})$. If the honest parties output $\perp$ in the execution or if $w \not\equiv_{\ell+s} 0$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{CorrectMult}}$ and halts. If $d \not\equiv_{\ell} 0$ and the honest parties did not abort, then $\mathcal{S}$ output fail and halts. Otherwise, it records $[\![a]\!]^{\mathcal{C}}_{\ell}, [\![b]\!]^{\mathcal{C}}_{\ell}, [\![c]\!]^{\mathcal{C}}_{\ell}$ as the output of the corrupted parties from this step.
2. Use the generated triple:
   (a) The simulator $\mathcal{S}$ receives the adversary's shares $[\![x]\!]^{\mathcal{C}}_{\ell}$ and $[\![y]\!]^{\mathcal{C}}_{\ell}$ from $\mathcal{F}_{\text{CorrectMult}}$. Then, $\mathcal{S}$ computes $[\![u]\!]^{\mathcal{C}}_{\ell} = [\![x]\!]^{\mathcal{C}}_{\ell} - [\![a]\!]^{\mathcal{C}}_{\ell}$ and $[\![v]\!]^{\mathcal{C}}_{\ell} = [\![y]\!]^{\mathcal{C}} - [\![b]\!]^{\mathcal{C}}$. Finally, $\mathcal{S}$ chooses random $u, v \in \mathbb{Z}_{2^\ell}$ and defines the honest parties' shares $[\![u]\!]^{H}_{\ell}$ and $[\![v]\!]^{H}_{\ell}$, by running $\mathsf{share}(u, [\![u]\!]^{\mathcal{C}}_{\ell})$ and $\mathsf{share}(v, [\![v]\!]^{\mathcal{C}}_{\ell})$ respectively.
   (b) $\mathcal{S}$ plays the role of the honest parties in the execution of $\mathsf{open}([\![u]\!]_{\ell})$ and $\mathsf{open}([\![v]\!]_{\ell})$. If the honest parties output $\perp$, then it sends abort to $\mathcal{F}_{\text{CorrectMult}}$ and halts.

(c) The simulator $\mathcal{S}$ defines the adversary's shares by the equation $[\![z]\!]^{\mathcal{C}}_{\ell} = [\![c]\!]^{\mathcal{C}}_{\ell} + u \cdot [\![b]\!]^{\mathcal{C}}_{\ell} + v \cdot [\![a]\!]^{\mathcal{C}}_{\ell} + u \cdot v$ and sends these to $\mathcal{F}_{\text{CorrectMult}}$.

Observe that given that the event that $\mathcal{S}$ outputs fail does not occur, the only difference between the simulation and the real execution is the way the values a'', u and v are set. In the simulation, these are randomly and independently sampled by $\mathcal{S}$. In contrast, in the real execution we have that $a'' = r \cdot a - a'$, $u = x - a$ and $v = y - b$. However, from the way $\mathcal{F}_{\mathsf{rand}}$ is defined, we have that a', a and b are guaranteed to be uniformly and independently distributed over the corresponding ring and thus so are a'', u and v. Thus, the adversary's view is identically distributed in the two executions (given that the fail output event does not happen).

Next, we show that given the identical view, the output of the honest parties is also identical in both executions. In the simulation, the honest parties' output is random shares of $x \cdot y$ given the corrupted parties' shares. In contrast, in the real execution, these are determined by computing $[\![z]\!]^{H}_{\ell} = [\![c]\!]^{H}_{\ell} + u \cdot [\![b]\!]^{H}_{\ell} + v \cdot [\![a]\!]^{H}_{\ell} + u \cdot v$. However, since $z = x \cdot y$ this obtained shares are random shares of $x \cdot y$ as in the simulation.

We conclude that the only difference between the executions is the fail event. However, by Lemma 3.4, this event happens with probability of at most 2^{-s}, which is exactly the statsitcal error allowed by the proposition. ∎

A.2 Proof of Proposition 3.6

Proposition A.2 (Proposition 3.6 - restated). *Protocol 3.7 securely computes functionality $\mathcal{F}_{\text{randBit}}$ with abort in the $(\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\text{CorrectMult}})$-hybrid model in the presence of malicious adversaries controlling $t < n/2$ parties.*

Proof: First, observe that simulation here is straightforward. Since the protocol has no inputs, the simulator $\mathcal{S}$ can perfectly simulate the honest parties in the execution (including aborting the protocol if the honest parties output $\perp$ when running the open procedure). In addition, $\mathcal{S}$ receives the corrupted parties' shares when playing the role of $\mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}_{\text{CorrectMult}}$ and thus it can compute locally $[\![b]\!]^{\mathcal{C}}_{\ell}$ and hand it to $\mathcal{F}_{\text{randBit}}$.

Next, we show that the honest parties' output is identically distributed in both the real and ideal executions. In the simulation, the honest parties' ouptut is random shares of a random bit (computed given the corrupted parties' shares). We now show that this is the same for the real world execution.

To see this, first observe that $c \equiv_{\ell+2} a^2$ (with no additive errors), since $\mathcal{F}_{\text{CorrectMult}}$ was used. Furthermore, using Lemma 4.1 in [DEF+19], we obtain that $d = \sqrt{c}^{-1} \cdot a \bmod 2^{\ell+2}$ satisfies $d \in \{\pm 1, \pm 1 + 2^{\ell+1}\}$, so in particular $d \equiv_{\ell+1} \pm 1$, with each one of these cases happening with equal probability. This implies that $b = b'/2 \bmod 2^{\ell}$ satisfies $b \equiv_{\ell} 0$ or $b \equiv_{\ell} 1$, each case with the same probability.

The final observation is that all the shares of $b' = d + 1 \bmod 2^{\ell+1}$ are even, which is required to ensure that the parties can execute the right-shift operation

in step 5. This is implied by the following argument. First of all, notice that $[\![d]\!]_{\ell+2} + 1 = 2 \cdot \sqrt{c}^{-1}[\![r]\!]_{\ell+2} + (\sqrt{c}^{-1} + 1)$. Now, the shares $2 \cdot \sqrt{c}^{-1}[\![r]\!]_{\ell+2}$ are even since these are obtained by multiplying the constant 2. Furthermore, the constant $(\sqrt{c}^{-1} + 1)$ is even since $\sqrt{c}^{-1}$ is odd, and by the assumptions of the secret sharing scheme each canonical share of it is either 0 or the constant itself (see the "shares of a constant" property in Section 2.1), so in particular all of its shares are even.

The above implies that at the end of the protocol, the parties hold a sharing of a random bit, exactly as in the simulation. This concludes the proof. ∎

A.3 Proof of Proposition 3.9

Proposition A.3 (Proposition 3.9 - restated). *Protocol 3.8 securely computes $\mathcal{F}_{\text{CheckZero}}$ with abort in the $(\mathcal{F}_{\text{randBit}}, \mathcal{F}_{\text{CorrectMult}})$-hybrid model in the presence of malicious adversaries who control $t < n/2$ parties.*

Proof: The simulation begins with the ideal world simulator $\mathcal{S}$ receiving the corrupted parties' shares $[\![v]\!]^{\mathcal{C}}_{\ell}$ and the output (accept or reject) from $\mathcal{F}_{\text{CheckZero}}$. Then, $\mathcal{S}$ works as follows:

1. Playing the role of $\mathcal{F}_{\text{randBit}}$, $\mathcal{S}$ receives $[\![r_0]\!]^{\mathcal{C}}_{\ell}, \ldots, [\![r_{\ell-1}]\!]^{\mathcal{C}}_{\ell}$ from $\mathcal{A}$.
2. $\mathcal{S}$ locally computes $[\![r]\!]^{\mathcal{C}}_{\ell} = \sum_{i=0}^{\ell-1}[\![r_i]\!]^{\mathcal{C}}_{\ell}$ and $[\![c]\!]^{\mathcal{C}}_{\ell} = [\![v]\!]^{\mathcal{C}}_{\ell} + [\![r]\!]^{\mathcal{C}}_{\ell}$. Then, it chooses a random $c \in \mathbb{Z}_{2^\ell}$ and computes $[\![c]\!]^{H}_{\ell} = \mathsf{share}(c, [\![c]\!]^{\mathcal{C}}_{\ell})$.
3. $\mathcal{S}$ simulates the execution of $\mathsf{open}([\![c]\!]_{\ell})$ by playing the role of the honest parties. If the honest parties output $\perp$ at the end of the execution, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{CheckZero}}$ and halts.
4. $\mathcal{S}$ locally converts $[\![r_i]\!]^{\mathcal{C}}_{\ell} \to [\![r_i]\!]^{\mathcal{C}}_{1}$ for $i = 1$ to ℓ.
5. $\mathcal{S}$ simulates the computation of the circuit by playing the role of $\mathcal{F}_{\text{CorrectMult}}(1)$. Let $[\![T]\!]_1$ be the sharing of the output of the circuit. Thus, $\mathcal{S}$ holds the corrupted parties' shares of the ouptut $[\![T]\!]^{\mathcal{C}}_{1}$.
6. If $\mathcal{S}$ received accept from $\mathcal{F}_{\text{CheckZero}}$, then it sets $b = 0$. Otherwise, in the case where $\mathcal{S}$ received reject from $\mathcal{F}_{\text{CheckZero}}$, it sets $b = 1$. Then, it runs $\mathsf{share}(b, [\![T]\!]^{\mathcal{C}}_{1})$ to obtain the honest parties' shares $[\![b]\!]^{H}_{1}$.
7. Finally, $\mathcal{S}$ simulates the opening of the output by playing the role of the honest parties. If the honest parties output $\perp$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{CheckZero}}$. Otherwise, it sends continue to $\mathcal{F}_{\text{CheckZero}}$.
8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

The difference between the simulation and the real execution is in the way c and the output of circuit b are computed. However, since $r \in \mathbb{Z}_{2^\ell}$ is secret and uniformly random, the opened value $c = v + r \mod 2^\ell$ is also uniformly distributed over the ring and thus it is identically distributed in both executions. Furthermore, $v \equiv_\ell 0$ if and only if $v + r \equiv_\ell r$, which is equivalent to the bit decomposition of c, $(c_0, \ldots, c_{\ell-1})$, being equal to that of r, $(r_0, \ldots, r_{\ell-1})$. Checking this is equivalent to checking that all the bits of $(r_0 \oplus c_0, \ldots, r_{\ell-1} \oplus c_{\ell-1})$ are zero, which is equivalent to $\bigvee_{i=0}^{\ell-1}(r_i \oplus c_i) = 0$. Thus, the value of b in the simulation, as chosen by $\mathcal{S}$, is exactly as in the real execution. This concludes the proof. ∎

B Proof of Security for the Main Protocol

Theorem B.1 (Theorem 4.3 - restated). *Let f be an n-party functionality over $\mathbb{Z}_{2^k}$ and let s be a statistical security parameter. Then, Protocol 4.1 securely computes f with abort in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\text{CheckZero}})$-hybrid model with statistical error $2^{-s+\log(s+1)}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Proof: Let $\mathcal{A}$ be the real world adversary who controls a set of corrupted parties $\mathcal{C}$ and let $\mathcal{S}$ be the ideal world simulator. The simulator $\mathcal{S}$ works as follows:

1. *Secret sharing the inputs:* $\mathcal{S}$ receives from $\mathcal{A}$ the set of corrupted parties inputs (values v_j associated with parties $P_i \in \mathcal{C}$) and the corrupted parties' shares $\{[\![v_m]\!]^{\mathcal{C}}_{k+s}\}_{m=1}^{M}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{input}}$ in the protocol.
2. *Generate the randomizing share:* Simulator $\mathcal{S}$ receives the share $[\![r]\!]^{\mathcal{C}}_{k+s}$ of the corrupted parties that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{rand}}$.
3. *Randomization of inputs:* For every input wire $m = 1, \ldots, M$, simulator $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$ in the multiplication of the mth input $[\![v_m]\!]_{k+s}$ with $[\![r]\!]_{k+s}$. Specifically, $\mathcal{S}$ hands $\mathcal{A}$ the corrupted parties shares in $[\![v_m]\!]_{k+s}$ and $[\![r]\!]_{k+s}$ (it has these shares from the previous steps). Next, $\mathcal{S}$ receives the additive value $d = \xi_m \in \mathbb{Z}_{2^{k+s}}$ and the corrupted parties' shares $[\![z]\!]^{\mathcal{C}}_{k+s}$ of the result that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{mult}}$. Simulator $\mathcal{S}$ stores all of these corrupted parties shares.
4. *Circuit emulation:* Throughout the emulation, $\mathcal{S}$ will use the fact that it knows the corrupted parties' shares on the input wires of the gate being computed. This holds initially from the steps above, and we will show it computes the output wires of each gate below. For each gate G_ℓ in the circuit,
   - *If G_ℓ is an addition gate:* Given the shares of the corrupted parties on the input wires, $\mathcal{S}$ locally adds them as specified by the protocol, and stores them.
   - *If G_ℓ is a multiplication-by-a-constant gate:* Given the shares of the corrupted parties on the input wire, $\mathcal{S}$ locally multiplies them by the constant as specified by the protocol, and stores them.
   - *If G_ℓ is a multiplication gate:* $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$ in this step (as in the randomization of inputs above). Specifically, simulator $\mathcal{S}$ hands $\mathcal{A}$ the corrupted parties' shares on the input wires as it expects to receive from $\mathcal{F}_{\mathsf{mult}}$ (it has these shares by the invariant), and receives from $\mathcal{A}$ the additive value as well as the corrupted parties' shares for the output. These additive values are $\delta_\ell \in \mathbb{Z}_{2^{k+s}}$ (for the multiplication of the actual values) and $\gamma_\ell \in \mathbb{Z}_{2^{k+s}}$ (for the multiplication of the randomized value), as defined in the proof of Lemma 4.2. $\mathcal{S}$ stores the corrupted parties' shares.
5. *Verification stage:* Simulator $\mathcal{S}$ works as follows. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{rand}}$ receiving the shares $[\![\alpha_1]\!]^{\mathcal{C}}_{k+s}, \ldots, [\![\alpha_N]\!]^{\mathcal{C}}_{k+s}$, $[\![\beta_1]\!]^{\mathcal{C}}_{k+s}, \ldots, [\![\beta_M]\!]^{\mathcal{C}}_{k+s} \in \mathbb{Z}_{2^{k+s}}$ sent to $\mathcal{F}_{\mathsf{rand}}$ by $\mathcal{A}$. Then, it plays the role of $\mathcal{F}_{\mathsf{DotProduct}}$ receiving the shares $[\![u]\!]^{\mathcal{C}}_{k+s}$, the shares $[\![w]\!]^{\mathcal{C}}_{k+s}$ and the additive attacks Θ_1 and Θ_2 sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{DotProduct}}$. Next, $\mathcal{S}$ chooses a random $r \in \mathbb{Z}_{2^{k+s}}$ and computes the shares of

r by $(r_1, \ldots, r_n) = \mathsf{share}(r, [\![r]\!]^{\mathcal{C}}_{k+s})$, using the shares $[\![r]\!]^{\mathcal{C}}_{k+s}$ provided by $\mathcal{A}$ in the "generate randomizing share" step above. Next, $\mathcal{S}$ simulates the honest parties sending their shares in $\mathsf{open}([\![r]\!]_{k+s})$ to $\mathcal{A}$, and receives the shares that $\mathcal{A}$ sends to the honest parties in this open. If any honest party would abort (it knows whether this would happen since it has all the honest parties' shares), then $\mathcal{S}$ simulates it sending $\perp$ to all parties, externally sends abort_j for every $P_j \in H$ to the trusted party computing f, and halts.
Finally, $\mathcal{S}$ simulates $\mathcal{F}_{\text{CheckZero}}$, as follows:

(a) If any non-zero $\xi_m, \delta_i, \gamma_i$ was provided to $\mathcal{F}_{\mathsf{mult}}$ by $\mathcal{A}$ in the simulation that is not congruent to 0 modulo 2^k, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{CheckZero}}$ sending reject to the parties.

(b) Otherwise, if any $\xi_m, \delta_i, \gamma_i$ was provided to $\mathcal{F}_{\mathsf{mult}}$ by $\mathcal{A}$ in the simulation that is not congruent to 0 modulo 2^{k+s}, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{CheckZero}}$ sending accept to the parties with probability p and reject with probability $1-p$, where $p = 2^{-(k+s-u)}$ and u is determined as follows:
If $\exists \xi_m \not\equiv_{k+s} 0$, let m_0 be the smallest m for which this holds. Then, u is the largest for which 2^u divides ξ_{m_0}.
Otherwise, all $\xi_m \equiv_{k+s} 0$, but there is $i \in \{1, \ldots, N\}$ for which $\delta_i \not\equiv_{k+s} 0$ or $\gamma_i \not\equiv_{k+s} 0$. Let i_0 be the smallest index for which this holds. Then, u is the largest for which 2^u divides $\gamma_{i_0} - r \cdot \delta_{i_0}$.

(c) Finally, if all $\xi_m, \delta_i, \gamma_i \equiv_{k+s} 0$, then $\mathcal{S}$ set $T = \Theta_1 - r \cdot \Theta_2$ and simulates $\mathcal{F}_{\text{CheckZero}}$ sending accept if $T \equiv_{k+s} 0$ and reject otherwise.

In any of the above, if $\mathcal{S}$ sent abort to $\mathcal{A}$, then $\mathcal{S}$ externally sends abort_j for every $P_j \in H$ to the trusted party computing f. Otherwise, $\mathcal{S}$ proceeds to the next step.

6. *Output reconstruction:* If no abort had occurred, $\mathcal{S}$ externally sends the trusted party computing f the corrupted parties' inputs that it received in the "secret sharing the inputs" step above. $\mathcal{S}$ receives back the output values for each output wire associated with a corrupted party. Then, $\mathcal{S}$ simulates the honest parties in the reconstruction of the corrupted parties' outputs. It does this by computing the shares of the honest parties on this wire using the corrupted parties' shares on the wire (which it has by the invariant) and the actual output value it received from the trusted party.
In addition, $\mathcal{S}$ receives the messages from $\mathcal{A}$ for the reconstructions to the honest parties. If any of the messages in the reconstruction of an output wire associated with an honest P_j are incorrect (i.e., the shares sent by $\mathcal{A}$ are not the correct shares it holds), then $\mathcal{S}$ sends abort_j to instruct the trusted party to not send the output to P_j. Otherwise, $\mathcal{S}$ sends $\mathsf{continue}_j$ to the trusted party, instructing it to send P_j its output.

We claim that the view of the adversary in the simulation is distributed identically to its view in the real execution, except with probability $1/2^{s-\log(s+1)}$. In order to see this, observe first that if all $\xi_m, \delta_i, \gamma_i$ values are congruent to 0 modulo 2^{k+s}, then the simulation is perfect.

Next, consider the case that some $\xi_m, \delta_i, \gamma_i$ value is not congruent to 0 modulo 2^k. In this case, the simulator $\mathcal{S}$ *always* simulates $\mathcal{F}_{\text{CheckZero}}$ outputting reject.

However, in a real execution where some $\xi_m, \delta_i, \gamma_i$ value is not congruent to 0 modulo 2^k, functionality $\mathcal{F}_{\text{CheckZero}}$ may return accept. This event happens when $T \equiv_{k+s} 0$. By Lemma 4.2, the probability that $T \equiv_{k+s} 0$ in such a real execution is less than $2^{-s+\log(s+1)}$. Thus, in this case, the statistical difference between these distributions is less than $2^{-s+\log(s+1)}$, as stated in the theorem.

Finally, we show that when all $\xi_m, \delta_i, \gamma_i$ are congruent to 0 modulo 2^k but *not* modulo 2^{k+s} the simulation is identically distributed to the real execution. Let $\Delta_i = \epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i$. If there exists $\xi_m \not\equiv_{k+s} 0$, then let m_0 be the smallest m for which this holds. Then, using Eq. (1), we have that $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ in the real execution if and only if

$$\beta_{m_0} \cdot \xi_{m_0} \equiv_{k+s} \left(-\sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).$$

Let 2^u be the largest power of 2 dividing ξ_{m_0}. Then we have

$$\beta_{m_0} \equiv_{k+s-u} \left(\frac{-\sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2)}{2^u} \right) \cdot \left(\frac{\xi_{m_0}}{2^u} \right)^{-1}.$$

Since β_{m_0} is chosen uniformly from $\mathbb{Z}_{2^{k+s}}$ and its value is kept secret, we obtain that $\mathcal{F}_{\text{CheckZero}}$ will return accept in the real execution with probability 2^{-k+s-u} which is exactly the probability that $\mathcal{S}$ sends accept to $\mathcal{A}$.

Otherwise, all $\xi_m \equiv_{k+s} 0$, but there exists i such that $\delta_i \not\equiv_{k+s} 0$ or $\gamma_i \not\equiv_{k+s} 0$. Let i_0 be the smallest i for which this holds. Observe that this implies that $\epsilon_{i_0} \equiv_{k+s} 0$, as there were no attacks on previous gates. Thus, we have in the real execution that $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ if and only if

$$\alpha_{i_0} \cdot (\gamma_{i_0} - r \cdot \delta_{i_0}) \equiv_{k+s} \left(-\sum_{\substack{i=1 \\ i \neq i_0}}^{N} \alpha_i \cdot \Delta_i - (\Theta_1 - r \cdot \Theta_2) \right).$$

Let u be the largest for which 2^u divides $\gamma_{i_0} - r \cdot \delta_{i_0}$. Then, we have that the above holds if and only if

$$\alpha_{i_0} \equiv_{k+s-u} \left(\frac{-\sum_{\substack{i=1 \\ i \neq i_0}}^{N} \alpha_i \cdot \Delta_i - (\Theta_1 - r \cdot \Theta_2)}{2^u} \right) \cdot \left(\frac{\gamma_{i_0} - r \cdot \delta_{i_0}}{2^u} \right)^{-1}.$$

As before, since α_{i_0} is distributed uniformly over $\mathbb{Z}_{2^{k+s}}$ and kept secret during the execution, the above holds with probability 2^{-k+s-u}. This is exactly the probability that $\mathcal{F}_{\text{CheckZero}}$, simulated by $\mathcal{S}$, outputs accept in the simulation.

Going over all cases, we conclude that the statistical difference between $\mathcal{A}$'s view in the real and simulated execution is $2^{-s+\log(s+1)}$. This concludes the proof. ∎

C Replicated Secret Sharing for Three Parties

We now present in detail the efficient three party instantiation of our compiler from replicated secret sharing. Sharing a value $x \in \mathbb{Z}_{2^\ell}$ is done by picking at random $x_1, x_2, x_3 \in \mathbb{Z}_{2^\ell}$ such that $\sum_i x_i \equiv_\ell x$. P_i's share of x is the pair (x_i, x_{i+1}) and we use the convention that $i+1 = 1$ when $i = 3$. To reconstruct a secret, P_i receives the missing share from the two other parties. Note that reconstructing a secret is robust in the sense that parties either reconstruct the correct value x or they abort.

Replicated secret sharing satisfies the properties described in Section 2.1, and one can efficiently realize the required functionalities described in the same section. Below we discuss some of these properties/functionalities.

Generating Random Shares. Shares of a random value can be generated non-interactively, as noted in [LN17,MR18], by making use of a setup phase in which each party P_i obtains shares of two random keys k_i, k_{i+1} for a pseudorandom function (PRF) F. The parties generate shares of a random value for the j-th time by letting P_i's share to be (r_i, r_{i+1}), where $r_i = F_{k_i}(j)$. These are replicated shares of the (pseudo)random value $r = \sum_i F_{k_i}(j)$. Proving that this securely computes $\mathcal{F}_{\mathsf{rand}}$ is straight forward and we omit the details.

Secure Multiplication up to an Additive Attack. To multiply two shared values, we use the protocol from [MR18,AFL+16], which is described in C.1. The shares of 0 that this protocol needs can be obtained by using correlated keys for a PRF, in similar fashion to the protocol for $\mathcal{F}_{\mathsf{rand}}$ sketched above.

PROTOCOL C.1 (**Secure multiplication up to an additive attack.**)

- **Inputs:** Parties hold sharings $[\![x]\!]$, $[\![y]\!]$ and additive sharings $(\alpha_1, \alpha_2, \alpha_3)$ where $\sum_{i=1}^{3} \alpha_i = 0$.
- **Protocol:**
  1. P_i computes $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1} + \alpha_i$ and sends z_i to P_{i-1}.
  2. P_j, upon receiving z_{j+1}, defines its share of $[\![x \cdot y]\!]$ as (z_j, z_{j+1}).

The above protocol is secure up to an additive attack as noted in [LN17]. We note that this can be extended to instantiate $\mathcal{F}_{\mathsf{DotProduct}}$ at the communication cost of one single multiplication, as shown in [CGH+18].

Efficient Checking Equality to 0. Checking that a value is a share of 0 can be performed very efficiently in this setting by relying on a Random Oracle $\mathcal{H}$. The observation we rely on is that, if $\sum_i x_i \equiv_\ell 0$, then $x_{i-1} \equiv_\ell -(x_i + x_{i+1})$ and

so P_i can send $z_i = \mathcal{H}(-(x_i + x_{i+1}))$ which will be equal to x_{i-1} which is held by P_{i+1} and P_{i-1}. Since only one party is corrupted, it suffices that each P_i will send it only to P_{i+1}. Upon receiving z_i from P_i, P_{i+1} checks that $z_i = \mathcal{H}(x_{i-1})$ and aborts if this is not the case.

This protocol is formalized in Protocol C.3 in the $\mathcal{F}_{\text{RO}}$-hybrid model. The $\mathcal{F}_{\text{RO}}$ functionality is described in Functionality C.2.

We remark that that this protocol does not instantiate $\mathcal{F}_{\text{CheckZero}}$ exactly. In order for the proof of security to work, we need to allow the adversary to cause the parties to reject also when $v = 0$. We denote this modified functionality by $\mathcal{F}_{\text{CheckZero}}{}'$. This is minor change since the main requirement from $\mathcal{F}_{\text{CheckZero}}$ in our compiler is that the parties won't accept a value as 0 when it is not, which is still satisfied by the modified functionality.

FUNCTIONALITY C.2 ($\mathcal{F}_{\text{RO}}$ – Random Oracle functionality)
Let M be an initially empty map.

- On input x from a party P, if $(x, y) \in M$ for some y, return y. Otherwise pick y at random and set $M = \{(x, y)\} \cup M$ and return y.
- On (x, y) from $\mathcal{S}$ and if $(x, \cdot) \notin M$ set $M = \{(x, y)\} \cup M$.

PROTOCOL C.3 (Checking Equality to 0 in the $\mathcal{F}_{\text{RO}}$-Hybrid Model)

- **Inputs:** Parties hold a sharing $[\![v]\!]$.
- **Protocol:**
  1. Party P_i queries $\beta_i \leftarrow \mathcal{F}_{\text{RO}}(-(v_i + v_{i+1}))$ and sends β_i to P_{i+1}.
  2. Upon receiving β_{i-1} from P_{i-1}, each party P_i checks that $\beta_{i-1} = \mathcal{F}_{\text{RO}}(v_{i+1})$. If this is not the case, then P_i outputs reject. Otherwise, it outputs accept.

Proposition C.4. *Protocol C.3 securely computes $\mathcal{F}_{\text{CheckZero}}$ in the $\mathcal{F}_{\text{RO}}$-hybrid model in the presence of one malicious corrupted party.*

Proof: Let $\mathcal{A}$ be the real adversary who corrupts at most one party and $\mathcal{S}$ the ideal world simulator. Let P_i be the corrupted party. The simulation begins with $\mathcal{S}$ receiving the shares of P_i, i.e., (v_i, v_{i+1}). Then, $\mathcal{S}$ proceed as follows:

- If $\mathcal{S}$ receives accept from $\mathcal{F}_{\text{CheckZero}}{}'$, then it knows that $v \equiv_\ell 0$ and so it can compute the share $v_{i-1} = -(v_i + v_{i+1})$ and so it knows the honest parties' shares and can perfectly simulate the execution, while playing the role of $\mathcal{F}_{\text{RO}}$. If $\mathcal{A}$ cause the parites to reject by using different shares, then $\mathcal{S}$ sends reject to $\mathcal{F}_{\text{CheckZero}}{}'$.

– If $\mathcal{S}$ receives reject, then it chooses a random $v_{i-1} \in \mathbb{Z}_{2^\ell} \setminus \{-(v_i + v_{i+1})\}$ and defines the honest patries' shares accordingly. Then, it plays the role of $\mathcal{F}_{\text{RO}}$ simulating the remaining of the protocol. By the definition of $\mathcal{F}_{\text{RO}}$, the view of $\mathcal{A}$ is distributed identically in the simulated and the real execution.

■

D Proofs for Section 6 - Shamir-SS Instantiation

D.1 Proof of Lemma 6.2: Securely Computing $\mathcal{F}_{\text{rand}}$

Lemma D.1 (Lemma 6.2 - restated). *Protocol 6.1 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\text{rand}}$ for $[\![\cdot]\!]$ with statistical error of at most $2^{-\kappa}$ in the presence of a malicious adversary controlling $t < n/2$ parties.*

Proof: Let $\mathcal{A}$ be the real-world adversary. The simulator $\mathcal{S}$ interacts with $\mathcal{A}$ by simulating the honest parties in an execution of the protocol. In doing so, $\mathcal{S}$ obtains honest parties' shares $\langle r_1 \rangle_H, \ldots, \langle r_{n-\tau} \rangle_H$.

We distinguish three cases:

1. If at least one of the simulated honest parties aborts in any of the executions of Protocol 6.3, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{rand}}$.
2. If the checks pass but the honest parties' shares are inconsistent, $\mathcal{S}$ outputs fail. By Lemma 6.4 this only happens with probability at most $2^{-\kappa}$, allowed by the claim.
3. In the remaining case, the checks of Protocol 6.3 pass and the honest parties' shares are consistent. $\mathcal{S}$ calculates the corrupted parties' shares $\langle r_1 \rangle_{\mathcal{C}}, \ldots, \langle r_{n-\tau} \rangle_{\mathcal{C}}$ from the honest parties' shares, and sends them to $\mathcal{F}_{\text{rand}}$.

Before the invocation of $\mathcal{F}_{\text{rand}}$, the honest parties have no private inputs, hence $\mathcal{S}$ simulates them perfectly and $\mathcal{A}$'s view will be identical to the real execution. Thus, the simulated honest parties will abort in the ideal execution precisely when they would in the real execution.

The only thing it remains to prove is that if the parties do not abort, the output shares are identically distributed in the real and ideal executions. In particular, we need to prove that in the real execution, the *sharings* are independent and uniformly sampled from $\langle \cdot \rangle$.

Let $H \subseteq \mathcal{H}$ be a subset of honest parties of size $n - \tau$, and let $C := \{1, \ldots, n\} \setminus H$ denote its complement. Let A_H, A_C denote the submatrices of A corresponding to the columns indexed by H and C respectively. Let $\langle \mathbf{s}_H \rangle$ denote the vector $\langle s_i \rangle_{i \in H}$ of length $n - \tau$, and correspondingly $\langle \mathbf{s}_C \rangle := \langle \mathbf{s}_i \rangle_{i \in C}$. Then $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T = A_H \langle \mathbf{s}_H \rangle + A_C \langle \mathbf{s}_C \rangle$. Since $\langle \mathbf{s}_H \rangle$ is wholly generated by the honest parties, it consists of $n - \tau$ independent and uniformly random sharings of $\langle \cdot \rangle$. A_H is invertible (since A is hyperinvertible), hence we also have that $\langle \mathbf{s}_H \rangle$ consists of independent and uniformly random sharings. Adding a fixed $A_C \langle \mathbf{s}_C \rangle$ will not affect the distribution, hence the sharings $\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle$ are independent and uniformly random sharings. ■

D.2 Proof of Lemma 6.7: Securely Computing $\mathcal{F}_{\mathsf{mult}}$

Lemma D.2 (Lemma 6.7 - restated). *Protocol 6.6 securely computes $\mathcal{F}_{\mathsf{mult}}$ with statistical error $\leq 2^{-\kappa}$ in the $\mathcal{F}_{\mathsf{rand}}$-hybrid model in the presence of a malicious adversary controlling $t < n/2$ parties.*

Proof: Without loss of generality, assume $2\tau + 1 = n$ (recall that τ is the secret sharing threshold and not the number of corrupted parties, and so the proof still holds for any $t < n/2$).

For the offline phase, the simulator acts as in Lemma 6.2. By the proof, we have that $[\![r]\!]$ is a correct sharing. The sharing $[\![r']\!]_{(2\tau)}$ is not well-defined, because the adversary can change its mind about its shares at any time. However, the adversary always knows the additive error $r' - r$ that it introduces by changing its shares.

For the online phase, $\mathcal{S}$ simulates the honest parties towards $\mathcal{A}$.

We distinguish two cases:

- *Case 1: P_1 is not corrupt.* The simulated P_1 receives $\{u_i\}_{i\in\mathcal{C}}$ from $\mathcal{A}$. If it receives $\perp$ for any value u_i, it sends abort to $\mathcal{F}_{\mathsf{mult}}$ and simulates P_1 aborting. Otherwise, it calls $\mathcal{F}_{\mathsf{mult}}$ and receives $\{x_i\}_{i\in\mathcal{C}}, \{y_i\}_{i\in\mathcal{C}}$. For any $i \in \mathcal{C}$, since $\mathcal{S}$ knows x_i, y_i, r'_i, it may calculate $\delta_i = x_i y_i - r'_i$ and thus the value $\pi(\lambda_i \delta_i)$ the adversary is supposed to send if it behaves honestly. The simulator can therefore extract $d = \sum_{i\in\mathcal{C}} u_i - \pi(\lambda_i\delta_i)$. $\mathcal{S}$ does not know the true value of δ, however it may sample $\delta \leftarrow \mathbb{Z}_{2^\ell}$, send it to the corrupted parties, and calculate the corrupted parties' shares as $z_i = r_i + \delta + d$.
  It then simulates the broadcast of δ. If the broadcast aborts, $\mathcal{S}$ simulates the parties aborting and sends abort to $\mathcal{F}_{\mathsf{mult}}$. Otherwise, it sends $d, \{z_i\}_{i\in\mathcal{C}}$ to $\mathcal{F}_{\mathsf{mult}}$, and outputs whatever $\mathcal{A}$ outputs.
  In the ideal execution, $\mathcal{A}$ receives a random δ. It cannot distinguish this from the real value $x \cdot y - r$, since r is uniformly random and by privacy of the secret-sharing scheme it does not have any information on it.
- *Case 2: P_1 is corrupt.* $\mathcal{S}$ samples $[\![\delta]\!]_{(2\tau)} \leftarrow [\![\cdot]\!]_{(2\tau)}$. For $i \in \mathcal{H}$ it sends $u_i = \pi(\lambda_i\delta_i)$ to the corrupted P_1. The simulated honest parties receive an identical broadcasted value δ', otherwise the broadcast protocol aborts.
  Since $\mathcal{S}$ knows δ, it can extract $d := \delta' - \delta$, and calculate the corrupted parties' shares as $z_i = r_i + \delta'$. It then sends $d, \{z_i\}_{i\in\mathcal{C}}$ to $\mathcal{F}_{\mathsf{mult}}$, and it outputs whatever $\mathcal{A}$ outputs.

As mentioned above, the adversary cannot distinguish whether it is talking to a simulator or the real parties, hence its output will be identical.

In the ideal execution where no abort took place, the actual (non-simulated) parties receive their shares $\{z_i\}_{i\in\mathcal{H}}$ directly from $\mathcal{F}_{\mathsf{mult}}$. The shares are consistent and will reconstruct to the secret $z = x \cdot y + d$. In the ideal execution, the shares are generated by the probabilistic function $\mathsf{share}(z, \{z_i\}_{z\in\mathcal{C}})$, such that the shares are uniformly random subject to the constraints on the shares.[14] In the real execution, the shares also correspond to z. The sharing in the real execution is

[14] Depending on the privacy threshold the constraints may fully determine the shares.

calculated as $[\![r]\!] + \delta$, where $[\![r]\!]$ is a uniformly random sharing. Therefore, the outputs are identical in both executions. ■

E Experiments

The following appendix expands upon the evaluation and experiments we performed on two instantiations of our compiler. More precisely, and as mentioned earlier, we instantiate our compiler with a replicated secret sharing protocol for the case of $n = 3$ and with the Shamir secret sharing protocol over $\mathbb{Z}_{2^\ell}$ as presented in [ACD+19].

E.1 Implementation Details

Both instantiations are implemented in C++ and we will make the source code available under a free license. Recall that the majority of computation takes place over the ring $\mathbb{Z}_{2^\ell}$ where $\ell = k + s$ with k being the "computation" parameter and s the security parameter. Our implementation supports computation over $\mathbb{Z}_{2^\ell}$ for arbitrary ℓ by relying on `gmp`, however we provide specializations for the cases $\ell = 64$ and $\ell = 128$ (corresponding to $k = s = 32$ and $k = s = 64$, respectively). The specialization for $\ell = 64$ relies on standard fixed width 64-bit wide types while the specialization for $\ell = 128$ relies on the `unsigned int128` extension for GCC.[15]

All symmetric primitives (hashing and PRGs mainly) are handled by libsodium, and we refer to their documentation for details on which concrete schemes are used.

For the Galois-ring variant our implementation only supports the explicit ring $R = \mathbb{Z}_{2^\ell}[X]/(h(x))$ with $h(X) = X^4 + X + 1$. This ring supports $2^4 - 1 = 15$ parties and the act of hard-coding the irreducible polynomial allows us to implement multiplication and division in the ring using lookup tables. It is worth remarking that operations in $\mathsf{GR}(2^\ell, d)$ are more expensive than certain prime fields (in particular, Mersenne primes as the ones used in [CGH+18]). Our implementation can perform roughly 17 million multiplications in $\mathsf{GR}(2^{64}, 4)$ and 9 million multiplications in $\mathsf{GR}(2^{128}, 4)$ per second.

E.2 Experimental setup.

As we have access to the implementations from [CGH+18],[16] but not all of the ones in [EKO+19] (in particular, the ShareMind protocols are not open source) our experimental setup replicates that of [EKO+19] for the sake of getting the best possible comparison. Since the type of machines do used in [EKO+19] does not match those used in [CGH+18], we rerun the benchmarks of the latter to maintain uniformity accross all our benchmarks.

[15] https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html
[16] Code can be found at [biu17a] and [biu17b]

We use the AWS `c5.9xlarge` instances which have 36 virtual cores, 72gb of memory and a 10Gpbs network. We benchmark our protocols in three different settings: *Colocated* in which all machines are placed in the same data-center (Frankfurt), *Continent* in which parties are located in Frankfurt, London and Paris, respectively (i.e., on the same continent); and *World* in which parties are located in Frankfurt, Northern California and Tokyo. To get an idea of the latency induced in the different settings, Table 1 shows the RTT between the different locations.

	A	B	C
Colocated	Frankfurt	Frankfurt	Frankfurt
Continent	Frankfurt	London	Paris
World	Frankfurt	North Cali	Tokyo
	AB (ms)	AC (ms)	BC (ms)
Colocated	0.35 ±0.04	0.31 ±0.2	0.3 ±0.4
Continent	16.83 ±0.23	8.09 ±0.8	6.82 ±0.11
World	143.83 ±0.50	242.51 ±0.61	109.43 ±0.38

Table 1: Average RTT ± mdev. Measured using `ping` over 50 messages.

For benchmarks with $n > 3$ we distribute the excess parties evenly among the above locations ordered alphabetically. E.g., for $n = 5$ we place 2 parties in Frankfurt, 2 in London and 1 in Paris.

E.3 Results—the 3-party case

We begin by discussing our protocols in the three party setting, which include our Replicated SS instantiation from Section 5 and our Shamir SS instantiation from Section 6 using three parties. We remark that our Shamir SS instantiation, unlike the Replicated SS, is not implemented in its entirety,[17] as we will discuss in detail in Section E.4. However, for the three party case Shamir SS is generally outperformed by Replicated SS, a pattern that is also present in our setting. The goal of including this instantiation here is to support this claim.

Comparison with ring-based protocol. We compare our three party protocols with the protocols from [EKO+19], which are the only *implemented* three-party actively secure protocols for computation over $\mathbb{Z}_{2^\ell}$, to the best of our knowledge. As we already argued, since we do not have access to the implementations of all the variants treated in [EKO+19] we match instead the experimental

[17] Our Shamir implementation does not include the generic check described in Section 3.3, however for $n = 3$ a more efficient check as the one described in Section C can be used.

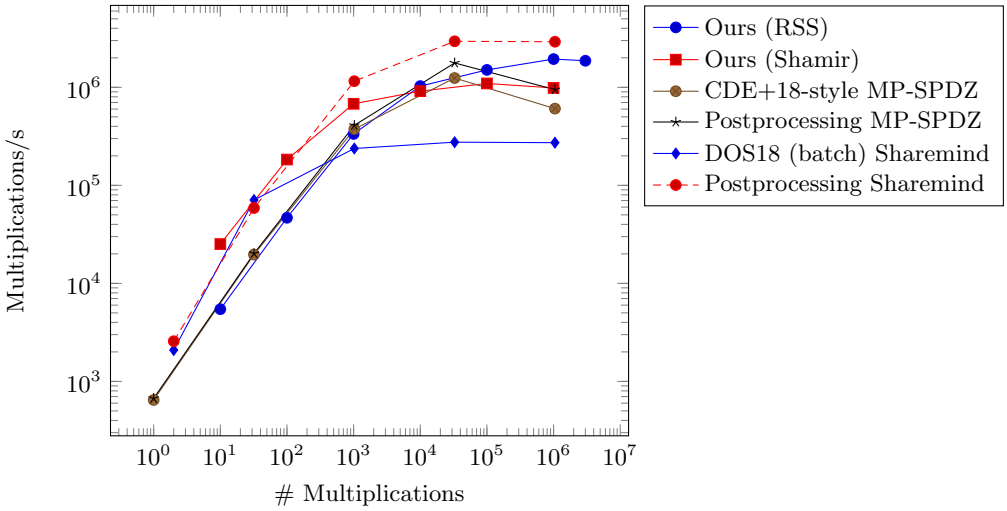

Fig. 1: Throughput in LAN

setup and make a direct comparison with the figures from [EKO+19]. The benchmarks here consist of measuring the throughput of computing a varying number of multiplications, and the results are included in Figures 1, 2 and 3. Unlike the numbers from [EKO+19], we experienced a linear tendency in the throughput all the way up to 1 million and so we increased the maximum batch size we used to 3 million.

The results from [EKO+19] use $k = 64$ and $s = 40$, and while we use the same k we use $s = 64$ as it is more efficient in our implementation. This provides stronger security guarantees but also requires us to send more data per multiplication (24 bits more, to be exact).

The results show that we outperform all the protocols of [EKO+19] except "Sharemind Postprocessing" in the LAN setting. We can attribute this to two things: First, our communication layer is sub-optimal in that a pairwise channel uses only one socket. In particular, a party cannot send and receive at the same time. For the communication pattern in RSS this means that one party must wait to receive a message before it can send its own—even if the two messages are unrelated. The second things is that both Sharemind and MP-SPDZ are more mature frameworks and as such probably have more efficient implementations of the arithmetic in $\mathbb{Z}_{2^\ell}$. It is worth noting that the communication pattern in the Shamir SS multiplication protocol does not run into this problem. For the two WAN settings we have a higher throughput which we attribute to the fact that we only need to send 2 $\mathbb{Z}_{2^\ell}$ elements per multiplication, while the Postprocessing protocols of [EKO+19] need to send 3.

We also observe that the Sharemind framework a commercial-grade implementation and therefore it is naturally more optimized.

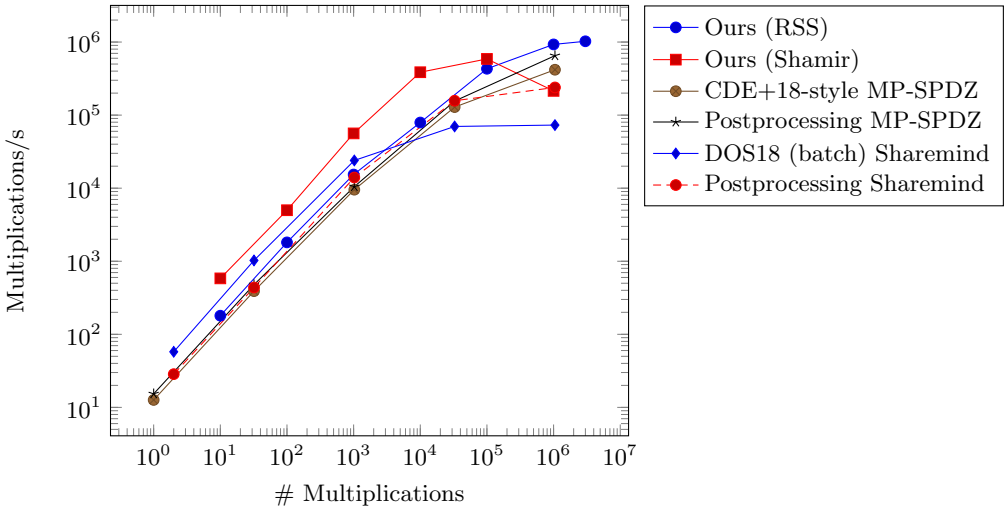

Fig. 2: Throughput in continental WAN

Comparison with ASTRA. Another recent protocol for computation over $\mathbb{Z}_{2^\ell}$ is [CCPS19]. However their implementation is not freely available and the benchmarks they run in the paper makes it hard to provide a good comparison. ASTRA only benchmark an AES computation which makes it hard to perform any meaningful comparison to our work.

Comparison with field-based protocol. Now we compare our three party instantiations against the 3-party protocols from [CGH+18]. Again, our compiler follows the template from that work, and as such our instantiations have many similarities as well with theirs, with the main difference being the algebraic structure over which they are defined.

We remark that the implementation from [CGH+18] uses a 61-bit Mersenne prime, which is useful for simplifying the modular reduction operation. This protocol supports simple integer arithmetic, bounded to 61-bits. Hence, it is natural to compare it with our implementation using $k = 64$, which provides roughly the same arithmetic capabilities. However, for more complex operations over fields like secure comparisons, the size of the underlying values must be bounded [CS10], and in this case it is natural to compare the 61-bit implementation from [CGH+18] with ours, using $k = 32$ and $s = 32$.

Our benchmarks bear similarities with the ones from [CGH+18]: We execute a series of circuits with 1 million multiplication gates with different depths, and measure the total running time. Results are shown in Table 2 and Table 3. Notice that for the WAN benchmarks we only run circuits of fairly low depth, as communication quickly becomes the bottleneck. We expect our protocol to perform better than the equivalent in [CGH+18] due to the fact that operations in $\mathbb{Z}_{2^\ell}$ can be implemented more efficiently—or at least as efficient—and indeed

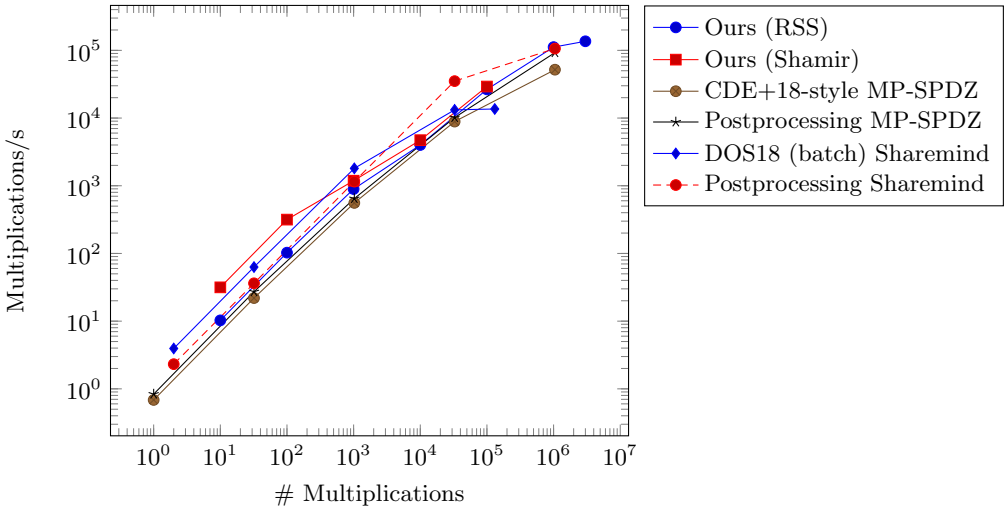

Fig. 3: Throughput in global WAN

this is what we see. Worth noting is that this does not hold for WAN setting, which can be explained using similar arguments as above; in particular, and extra message of communication matters more when the latency is larger. Additionally, our protocol also requires sending more data (essentially $2s$ more bits) per multiplication.

E.4 Results—the n-party case

Now we present our results for our Shamir SS instantiation in the more general multiparty case. First, we must point out that our implementation is *not complete*, as it omits the generic check-to-zero presented in Section 3.3. Since this step is executed only once during execution it is reasonable to assume that it adds a constant overhead, which will be negligible for larger circuits.

The only protocol for computation over rings in the honest majority setting is the one from [ACD+19], which forms the basis of the one considered in this work. However, their protocol is not implemented (and their focus was not concrete efficiency), so there is no ring-based protocol to compare against.

Instead, we compare our protocol to the Shamir protocol from [CGH+18], which is similar in structure to ours. We proceed in a similar fashion as the comparison to [CGH+18] discussed in the previous section. We make sure to time only the parts of the protocol from [CGH+18] for which we have an implementation; that is, the preparation phase (in which a number of double-shares are generated) and the online phase. Our results are shown in Table 4. We run the code from [CGH+18] in order to perform extract the relevant timings for the offline and online phase.

	Protocol	20	100	1,000	10,000
RSS	Ours $\ell = 64$	0.23	0.23	0.49	2.36
	Ours $\ell = 128$	0.4	0.41	0.56	2.47
	[CGH+18]	0.26	0.33	0.59	2.53
Shamir	Ours $\ell = 64$	3.07 (0.09)	2.99 (0.09)	3.28 (0.35)	4.26 (1.34)
	Ours $\ell = 128$	4.46 (0.24)	4.26 (0.22)	4.63 (0.54)	5.91 (1.79)
	[CGH+18]	0.61 (0.18)	0.84 (0.42)	1.37 (0.96)	7.69 (7.31)

Table 2: LAN times in seconds for circuits with 10^6 multiplications and varying depth with three parties. For the Shamir based protocols, the number in the parenthesis is the time spent in the online phase.

We can observe from Table 4 that our offline phase is substantially slower than the one from [CGH+18], but the online phase is actually competitive. Motivated by this observation, we also evaluate the throughput of the online phase of our protocol in a manner similar to the 3-party case presented in the previous section. The fast online phase of our protocol is mainly due to the fact that communication is very efficient since parties only need to send a single $\mathbb{Z}_{2^\ell}$ element to one party, who performs a broadcast (again with $\mathbb{Z}_{2^\ell}$) elements.

Finally, we include some throughput analysis of our double-share method in Tables 4 and 5. These are the bottleneck of our Shamir-based protocol, and these results shows the impact of this preprocessing in the throughput.

	Protocol	Continental		Global	
		20	**100**	**20**	**100**
RSS	Ours $\ell = 64$	0.81	1.32	8.53	15.89
	Ours $\ell = 128$	1.19	1.45	13.33	17.72
	[CGH+18]	0.54	1.14	5.9	14.32
Shamir	Ours $\ell = 64$	9.4 (0.87)	10.71 (1.48)	44.5 (10.52)	76.68 (43.09)
	Ours $\ell = 128$	11.41 (0.91)	12.42 (1.71)	74.94 (13.63)	105.03 (44.34)
	[CGH+18]	1.94 (0.92)	4.66 (3.84)	19.99 (11.48)	69.31 (59.12)

Table 3: WAN times in seconds for circuits with 10^6 multiplications and varying depth with three parties. For the Shamir based protocols, the number in the parenthesis is the time spent in the online phase.

Depth	**Protocol**	**3**	**5**	**7**	**9**
20	Ours $\ell = 64$	2.98 / 0.09	5.78 / 0.13	10.12 / 0.14	14.09 / 0.21
	Ours $\ell = 128$	4.23 / 0.24	7.70 / 0.29	12.60 / 0.33	15.50 / 0.38
	[CGH+18]	0.43 / 0.18	0.63 / 0.22	0.93 / 0.45	0.10 / 0.28
100	Ours $\ell = 64$	2.90 / 0.09	5.69 / 0.14	9.95 / 0.19	12.70 / 0.19
	Ours $\ell = 128$	4.04 / 0.22	7.72 / 0.32	12.55 / 0.41	15.30 / 0.41
	[CGH+18]	0.42 / 0.42	6.36 / 4.26	0.9 / 0.52	0.1 / 1.27
$1,000$	Ours $\ell = 64$	2.93 / 0.35	5.72 / 0.50	9.92 / 0.68	11.96 / 0.85
	Ours $\ell = 128$	4.09 / 0.54	7.66 / 0.81	12.50 / 1.06	15.27 / 1.18
	[CGH+18]	0.41 / 0.96	0.63 / 0.11	0.89 / 0.95	1.05 / 1.17
$10,000$	Ours $\ell = 64$	2.92 / 1.34	5.72 / 4.04	9.97 / 5.68	12.11 / 7.42
	Ours $\ell = 128$	4.12 / 1.79	7.69 / 3.85	12.53 / 5.91	15.39 / 7.34
	[CGH+18]	0.38 / 7.30	0.61 / 7.32	0.89 / 0.84	1.05 / 12.88

Table 4: LAN running times in seconds for circuits with 10^6 multiplications, different depth and for varying number of parties, evaluated using Shamir SS-based MPC. Each value is a tuple a/b where a is the preprocessing time (which is dominated by the double-share generation) and b is the time it takes to evaluate the circuit.

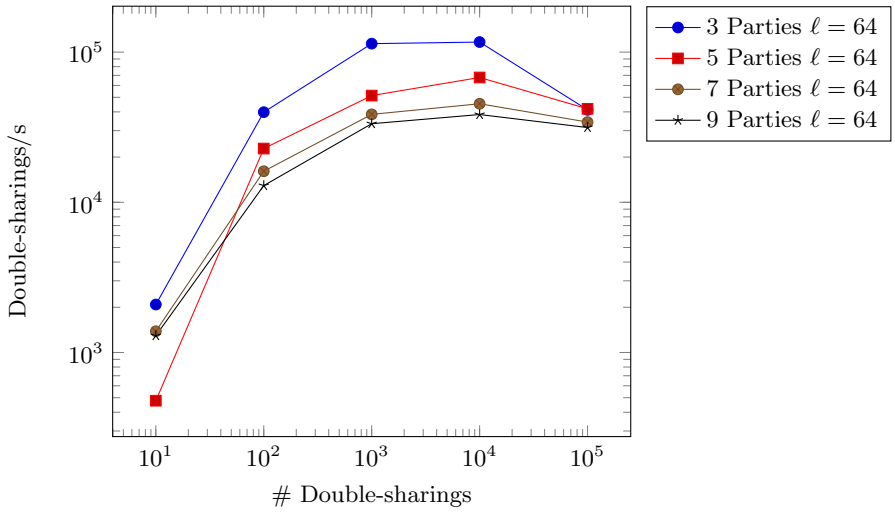

Fig. 4: Throughput in LAN of our double-share protocol for different parties, with $\ell = 64$

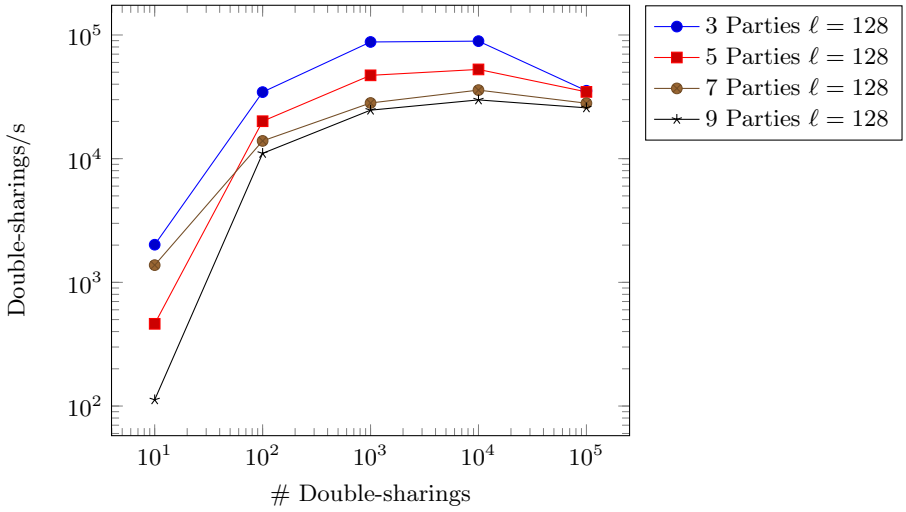

Fig. 5: Throughput in LAN of our double-share protocol for different parties, with $\ell = 128$