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

Abstract. Multiparty computation (MPC) over rings such as $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ has received a lot of attention recently due to the potential benefits in implementation and performance. Several actively secure protocols over these rings have been implemented, for both the dishonest majority setting and the setting of three parties with one corruption. However, in the honest majority setting, no concretely-efficient protocol for arithmetic computation over rings has yet been proposed that allows for an arbitrary number of parties.

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 compiles them into a maliciously secure protocol. At a high level, we follow the framework of Chida et al. (CRYPTO 2018) for finite fields, and use techniques from Cramer et al. (CRYPTO 2018) to achieve compatibility for rings with only a small 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.

Using our compiler, we obtain two maliciously secure protocols that are both highly efficient. The first works for three parties and is based on replicated secret sharing, and requires each party to send just $2(k+s)$ bits per multiplication gate. Our second protocol works for any number of parties, and uses a variant of Shamir secret sharing that was recently proposed by Abspoel et al. (TCC 2019). The protocol requires each party to send just $3(k+s)$ bits in the online computation and $3(k+s)\log n$ bits in the input-independent offline step, 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 an *arbitrary* number of parties. We implemented our two protocols, run extensive experiments to measure their performance and report their efficiency. Our results are the first to show experimentally that computation over rings for any n is practically viable.

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 intermediate values of the computation remain hidden. MPC has by now been studied for several decades, and different protocols have been developed throughout the years.

Most MPC protocols are "general purpose", meaning that they can in principle compute *any* computable 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) [5,6,20,32,26,16,33].

It was only 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 [17], while several other works focus on honest majority case for small number of parties [26,16,2,25]. Computation over $\mathbb{Z}_{2^k}$ is appealing due benefits in performance over computation over fields, as verified in [24]. These benefits are partly due to 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 [24,34]. Though results are recent, MPC over rings has already been used in applications like privacy-preserving machine learning and secure evaluation of neural networks [34,36,18,40]. However, to this date, no concretely-efficient protocol that works for any number of parties has been proposed in the honest majority setting.

1.1 Our Contributions

In this work, we develop highly efficient protocols over $\mathbb{Z}_{2^k}$ by presenting a generic compiler that transforms a passively secure protocol for computation over $\mathbb{Z}_{2^{k+s}}$, to a protocol over the ring $\mathbb{Z}_{2^k}$ which is actively secure with abort and provides roughly s bits of statistical security.[1] We require our input protocol to be secure up to additive attack: an active adversary attacking the protocol may only introduce an additive error at each multiplication gate, rather than arbitrary (e.g. multiplicative) errors. While this may seem like a strict condition, in fact most existing passively secure protocols satisfy this condition [28]. Both the input and output protocols of our compiler 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 property is crucial for many applications like secure array indexing [7], or even more importantly applications

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

relying on matrix arithmetic like SVMs or neural networks, as shown for example in [34], and has not been achieved over rings in the literature before.

We apply our compiler to two passively secure protocols over $\mathbb{Z}_{2^k}$, and thus obtain two actively secure 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 [2], which embeds the shares in an extension ring. Because the size of the shares of our secret-sharing scheme is increased by a factor of $\log n$ bits, each party needs to communicate $3(k+s)\log n$ bits in the input-independent offline step. Through a novel optimization that also applies to the passively secure protocol, we are able to eliminate the $\log n$ factor in the online computation, resulting with bandwidth of just $3(k+s)$ bits in the online computation.

The second protocol works in the three-party setting and is extremely efficient. It is based on a protocol that uses replicated secret sharing and which is known to be very efficient [4]. 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. Previous protocols like [25,26] use Beaver-based preprocessing and do not achieve this property.

Implementation. We compare both protocols to other works, both theoretically and empirically. Our implementation for the 3-party protocol shows that 1 million multiplications in a circuit with depth-20 over the ring $\mathbb{Z}_{2^{64}}$, can be processed in 400 ms in a LAN network. For the Shamir-based protocol, we are able to process a circuit of 1 million multiplications in depth-20 over the ring $\mathbb{Z}_{2^{64}}$ in 2 seconds overall with 3 parties in a LAN network, from which only 200 ms are spent on the online computation. For 9 parties, this increases to just 3 seconds, from which 500 ms are spent on the online computation To obtain these results, we implemented the secret-sharing scheme over $\mathbb{Z}_{2^k}$ via Galois rings of [2]. Our work demonstrates the practical viability of these techniques, which may have applications beyond MPC.

Our protocols perform well with respect to their field analogues [16], which illustrates the benefits of working over the ring $\mathbb{Z}_{2^k}$ in terms of concrete efficiency.

1.2 Overview of our Techniques

Compiler. The starting point of our work is the general compiler by Chida et al. [16], which achieves a similar result to ours, but over fields. In that work, the authors show that passively secure honest majority protocols over fields, that satisfy the property of being secure up to additive attacks, can be compiled to achieve active security without a noticeable loss in efficiency. The main idea behind their compiler is that each input and intermediate value is encoded as $x \mapsto (x, r \cdot x)$ with a random secret r. The computation is carried out over this

AMD encoding, with the property that adversary which adds an offset to the codeword that is independent of x and r, is detected with high probability. Over fields the statistical security grows with the field size, but this does not hold for non-field rings $\mathbb{Z}_{2^k}$. In particular, the probability of detecting errors for the ring $\mathbb{Z}_{2^k}$ remains $1/2$, regardless of the size of k.

Our main observation is that a similar issue appears in the dishonest majority scenario, where a MAC scheme is required in order to enforce correct behavior from the parties when reconstructing a value. More precisely, the MAC scheme used in the SPDZ protocol [23] relies on the same property as above. This MAC scheme was adapted to work over $\mathbb{Z}_{2^k}$ in [17] by, instead of working over $\mathbb{Z}_{2^k}$, moving 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 [16], and using the "SPD$\mathbb{Z}_{2^k}$ trick" from [17] for the underlying AMD code, and it is described and analyzed in Section 4. However, various subtle issues arise when trying to combine these two techniques directly.

For example, one of the critical steps in the compiler is to check whether a given secret-shared value x equals 0, without revealing anything else. Over fields, this can be done by opening $r \cdot x$ for uniformly random r; with high probability $rx = 0$ if and only if $x = 0$, and for $x \neq 0$ we have that rx is uniformly random. However, this does not work over $\mathbb{Z}_{2^k}$, e.g. since for $x = 2^{k-1}$ we have $rx = 0$ with probability $1/2$. In Section 3.1 we present a novel check for equality with zero that works over rings, which may be of independent interest.

Efficient input protocols. Similarly to [16], 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 5.1 that the proof presented in [33], which shows that replicated secret sharing over fields is secure up to additive attacks, also holds over $\mathbb{Z}_{2^k}$.

Now, for the second protocol, we rely on the recent work of [2] that extends Shamir secret sharing to the ring setting via Galois rings. Their protocol for the honest majority setting is based on [5], but since they aim for guaranteed output delivery, their techniques are quite complex and the concrete efficiency is not so clear. We show how to use their techniques in an efficient way. This includes protocols for efficiently generating random sharings and multiplication of shared values. In particular, we combine their core ideas with the semi-honest protocol from [21], to obtain a simple multiplication protocol that is secure against additive errors. Moreover, note that the secret-sharing scheme from [2] does not have shares in the secret space ring $\mathbb{Z}_{2^k}$ but in a Galois ring extension $R \cong (\mathbb{Z}_{2^k})^{\lceil \log(n) \rceil}$. Through a simple but novel optimization detailed in Section 5.2, we are able to avoid communicating full elements in R in the online phase, but instead communicate just one element of $\mathbb{Z}_{2^k}$, ensuring the online phase has linear complexity in the number of players.

1.3 Related Work

The only previous general compiler with concrete efficiency over rings, to the best of our knowledge, is the compiler of [22], which was improved by [25].

However, their compiler does not preserve the adversary threshold when moving from passive to active security. In addition, in [22] and [25] the compiler was instantiated for the 3-party case only.

The only concretely efficient protocol for arithmetic computation over rings that works for *any* number of parties is the SPD$\mathbb{Z}_{2^k}$ protocol [17] which was proven to be practical in [24]. 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 Sharemind protocol [8] is being used to solve real-world problems but provides only passive security. The actively secure protocol of [26], which was optimized and implemented in [3], 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 [15] has a slightly overall higher bandwidth than [3], but focuses on minimizing online (input-dependent) cost. The actively secure three-party protocol of [25] is the closest to our protocol in the sense that they also focus on efficiency for large rings. 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 [25] in Section 6.4. Finally, parallel to this work, a new promising direction was presented by [11]. They used the sub-linear distributed zero-knowledge proof from [9] to achieve malicious security at almost the same cost as semi-honest for large circuits. However, their protocol was not implemented and experiments that were carried-out for 31-bit Merssene field, show that only their verification step takes several seconds for a 1-million gate circuit due to heavy computational work. This is expected to be even worse for rings. Also, it is not clear at the moment how to extend their techniques to more than 3 parties.

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 Linear Secret Sharing and its Properties

Let ℓ be a positive integer. A perfect (t, n)-secret-sharing scheme (SSS) 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 denote by $\mathsf{share}(x)$ the sharing interactive procedure and by $\mathsf{open}([\![x]\!])$ the procedure to open a sharing and reveal the secret. The share procedure may take also in addition to x, a set of shares $\{x_i\}_{i\in J}$ for $J \subset [n]$ and $|J| \leq t$, such that $\mathsf{share}(x, \{x_i\}_{i\in J})$ satisfies $[\![x]\!] = (x'_1, \ldots, x'_n)$, with $x'_i = x_i$ for $i \in J$. The open procedure may take an index i as an additional input. In this case, the secret is revealed to P_i only. In case the sharing $[\![x]\!]$ is not correct as defined below, $\mathsf{open}([\![x]\!])$ will output $\perp$. An SSS is linear if it allows the parties to obtain shares of linear combinations of secret-shared values without interaction.

Our compiler applies to any linear SSS over $\mathbb{Z}_{2^k}$ that has a multiplication protocol that is secure against additive attacks, as defined in Section 2.3. The only extra, non-standard properties required by our compiler are the following (for a formalization of the requirements of the SSS, see Appendix A):

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 $x' = x/2$.[2]

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. 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 $\perp$). The formal definition appears in Appendix A.

2.2 Security Definition

We use the standard definition of security based on the ideal/real model paradigm [12,29], with security formalized for non-unanimous abort. This means that the

[2] 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.

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 [30]. 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 [27,28]

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 presence of a malicious adversary and then running a verification step that enables the honest parties to detect cheating. In order to achieve 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 1. 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 5.1 and 5.2), 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. [2], 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 1 $\mathcal{F}_{\mathsf{mult}}(\ell)$

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 consider a similar functionality $\mathcal{F}_{\mathsf{DotProduct}}$ that, instead of computing one single multiplication, 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 formalized in Functionality 2, in Appendix B. As in [16], we will show that the functionality can be realized at almost the same cost as $\mathcal{F}_{\mathsf{mult}}$.

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 3.1. 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 [4] 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 5.3), we consider an instantiation which relies on super-invertible matrices [21] to achieve linear communication complexity, together with the "tensoring-trick" from [13,2] 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.

$\mathcal{F}_{\mathbf{bc}}$ – 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 [39]. 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 (as mentioned in Section 2.2).

$\mathcal{F}_{\text{input}}$ – Secure Sharing of Inputs. In this section, we present our protocol for securely sharing the parties' inputs. The protocol is the same as in [16] (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 $[\![r + (x - r)]\!] = [\![x]\!]$. The functionality this protocol instantiates is denoted by $\mathcal{F}_{\text{input}}$, and it is formalized in Section B in the Appendix.

$\mathcal{F}_{\text{CheckZero}}$ – Checking Equality to 0. We assume a functionality $\mathcal{F}_{\text{CheckZero}}$, which receives $[\![v]\!]_\ell^H$ from the honest parties, uses them to compute v and sends accept to all parties if $v \equiv_\ell 0$. Else, if $v \not\equiv_\ell 0$, the functionality sends reject. This functionality is described in detail as Functionality 4 in Section C in the appendix. In the same section, we show how to instantiate this functionality, which is non-trivial when compared to the field sertting. For example, 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.

Instead of following the same idea as over fields, we adapt the ideas introduced in [24] to perform oblivious equality check. In a nutshell, this technique consists of generating shares of the bit representation of the shared value $[\![v]\!]_\ell$, and then executing an OR circuit to check that all these bits are 0. This approach requires several tools, like a correct multiplication protocol and the generation of shared values $[\![b]\!]_\ell$ where b is a random bit. Furthermore, the results in [24] are set in the setting of a concrete secret-sharing scheme, namely additive secret sharing. In Section C we show how to implement these requirements and how to extend the techniques of [24] to an arbitrary secret-sharing scheme.

Finally, although this instantiation is generic in the sense that it works for an arbitrary secret sharing scheme satisfying the conditions stated in Sections 2.1 and A, it is not very efficient. In practice, concrete secret sharing schemes allow for cheaper check-to-zero protocols, as show later on.

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 1. Our protocol follows the paradigm of [16] 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 5.1 and Section 5.2).

Protocol 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 $\mathcal{F}_{\mathrm{input}}(k+s)$.
   (b) Each party P_j records its vector of shares $(x_1^j, \ldots, x_M^j)$ of all inputs, as received from $\mathcal{F}_{\mathrm{input}}(k+s)$. If a party received $\perp$ from $\mathcal{F}_{\mathrm{input}}$, 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 $\mathcal{F}_{\mathrm{CheckZero}}(k+s)$ on $[\![T]\!]_{k+s}$. If $\mathcal{F}_{\mathrm{CheckZero}}(k+s)$ 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 [16], 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 $r \cdot x_{i_0}$ 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 [17] 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 1 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 1, 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 1. *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 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.[3] Thus, it holds

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

$$\begin{aligned}\mathsf{val}([\![u]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) \\ &\qquad + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \Theta_1 \\ \mathsf{val}([\![w]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2\end{aligned}$$

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

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

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

Our goal is to show that $\mathsf{val}([\![T]\!])^H$, as shown in Eq. (2), 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 $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ if and only if

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

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

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 $\mathbb{Z}_{2^{k+s}}$.

– *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, $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ 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:

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

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 1.

Theorem 1. *Let f be an n-party functionality over $\mathbb{Z}_{2^k}$ and let s be a statistical security parameter. Then, Protocol 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 1. $\mathcal{S}$ 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 $[\![\alpha_1]\!]^{\mathcal{C}}_{k+s}, \ldots, [\![\alpha_N]\!]^{\mathcal{C}}_{k+s}$ and $[\![\beta_1]\!]^{\mathcal{C}}_{k+s}, \ldots, [\![\beta_M]\!]^{\mathcal{C}}_{k+s} \in \mathbb{Z}_{2^{k+s}}$ 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 $\mathcal{F}_{\mathrm{CheckZero}}$, 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 $\mathcal{F}_{\mathrm{CheckZero}}$ 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 $\mathcal{F}_{\mathrm{CheckZero}}$ 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 $\mathcal{F}_{\mathrm{CheckZero}}$ 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 1, 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. (2), 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. ■

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 $\mathsf{open}([\![r]\!]_{k+s}, i)$ 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 $\mathsf{open}([\![r]\!]_{k+s})$ 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 $\mathsf{open}([\![v]\!]_k, i)$.

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 5.1, $\mathcal{F}_{\mathsf{rand}}$ 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 [14,34,24,19]. 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 [34,24]. In particular, the work of [24] 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 3.1 is already an instantiation of a basic primitive. Furthemore, just like we adapted this check-to-zero from [24] 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 C, 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 [24].

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 Two Protocols Secure up to Additive Attack

In this section, we present a brief overview of two protocols that are passively secure up to additive attack, and which we will compile to active security. Details can be found in the appendices.

5.1 Replicated Secret Sharing for Three Parties

The first protocol is an efficient three party instantiation of our compiler from replicated secret sharing. To share a value $x \in \mathbb{Z}_{2^\ell}$ choose uniformly random $x_1, x_2, x_3 \in \mathbb{Z}_{2^\ell}$ subject to the condition $\sum_i x_i \equiv_\ell x$. Each player P_i has the pair (x_i, x_{i+1}) as their share; we take subscript indices modulo 3, i.e. P_3 holds (x_3, x_1). 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 $\mathcal{F}_{\mathsf{rand}}$ can be realized with out any communication and $\mathcal{F}_{\mathsf{mult}}$ can be realized by having each party sending one ring element [34,4,33]. Also $\mathcal{F}_{\mathsf{DotProduct}}$ can be computed at the same cost of $\mathcal{F}_{\mathsf{mult}}$ [16].

In addition, $\mathcal{F}_{\mathrm{CheckZero}}$ can be realized very efficiently by relying on a random oracle $\mathcal{H}$, as follows. We want to check whether $\sum_i x_i \equiv_\ell 0$, or equivalently $x_{i-1} \equiv_\ell -(x_i + x_{i+1})$. Now, each P_i computes $z_i = \mathcal{H}(-(x_i + x_{i+1})) = \mathcal{H}(x_{i-1})$. Since x_{i-1} is held by P_{i-1} and P_{i+1}, they can also compute this value. Since only one party is corrupted, it suffices that each P_i will send z_i to P_{i+1}, and each receiving party verifies whether it matches the value they expect, and aborts if it does not. For completeness, we present all protocols in Appendix D.

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.

5.2 Shamir Secret Sharing for Any Number of Parties

In this section, we present our instantiation based on Shamir's secret sharing over rings, using the techniques from [2]. 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 [2] 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.[4] For more relevant structural properties of Galois rings, see [2].

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

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

the privacy parameter of the scheme. Then, the set of *correct* share vectors is

$$C_\tau = \left\{ (f(\alpha_1), \ldots, f(\alpha_n)) \in R^n \;\middle|\; \begin{matrix} f \in R[X],\ \deg(f) \leq \tau, \\ \text{and } f(\alpha_0) \in \mathbb{Z}_{2^\ell} \subset R \end{matrix} \right\}. \tag{3}$$

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 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 $[\![x \cdot y]\!]_{(2t)} = (x_1y_1, \ldots, x_ny_n)$ — 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 $[\![\cdot]\!]_{(2t)}$ 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.

5.3 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[5] Vandermonde matrix [21].[6] 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 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 $[\![\cdot]\!]$

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

[6] 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.

be interpreted as an R-linear secret-sharing scheme [13,2]. 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 column vector $([\![x_1]\!], \ldots, [\![x_d]\!])^T$. 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 $M_r([\![x_1]\!], \ldots, [\![x_d]\!])^T$. If a single party P has a vector of shares $(s_1, \ldots, s_d) \in R$ for $\langle x\rangle = ([\![x_1]\!], \ldots, [\![x_d]\!])^T$, 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 $\langle s_k\rangle = ([\![s_{k1}]\!], [\![s_{kd}]\!])^\intercal$. 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.[7] 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 [2].

Protocol 2 Generating random sharings of $[\![\cdot]\!]$

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 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 $\langle r_j\rangle = ([\![r_{j1}]\!], \ldots, [\![r_{jd}]\!])$ and output $[\![r_{11}]\!], \ldots, [\![r_{1d}]\!], [\![r_{21}]\!], \ldots, [\![r_{(n-\tau)d}]\!]$.

Lemma 2. *Protocol 2 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{rand}}$ for $[\![\cdot]\!]$ 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 E.1

5.4 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.

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

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.

Protocol 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 3. *If at least one of the input sharings $\langle x_1\rangle, \ldots, \langle x_N\rangle$ is incorrect, Protocol 3 outputs* correct *with probability at most $\frac{1}{2^d}$.*

To show correctness, we use the following consequence from [2, Lemma 3].

Lemma 4. *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 3] Let C denote the R-module of correct share vectors (such as in (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 4 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. ■

5.5 Secure Multiplication up to Additive Attacks

Multiplication follows the outline of the passively secure protocol of [21]. The protocol begins with an offline phase, where *random double sharings* are produced, i.e. a pair of sharings $([\![r]\!], [\![r]\!]_{(2t)})$ of the same uniformly random element r shared using polynomials of degree τ and degree 2τ, respectively.

We denote a double sharing as $[\![r]\!]_{(\tau,2\tau)} := ((r_1, r'_1), \ldots, (r_n, r'_n))$. 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

$$\left\{ ((f(\alpha_1), g(\alpha_1)), \ldots, (f(\alpha_n), g(\alpha_n))) \;\middle|\; \begin{array}{c} f, g \in R[X], \\ f(\alpha_0) = g(\alpha_0) \in \mathbb{Z}_{2^\ell}, \\ \deg(f) \leq \tau, \ \deg(g) \leq 2\tau \end{array} \right\}.$$

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

To generate sharings in $[\![\cdot]\!]_{(\tau,2\tau)}$, we essentially use Protocol 2. However, this protocol does not securely realize $\mathcal{F}_{\mathsf{rand}}$, since in Lemma 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 $[\![\cdot]\!]_{(2t)}$, 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 4 Secure multiplication up to an additive attack

- **Inputs:** Parties hold correct sharings $[\![x]\!]$, $[\![y]\!]$
- **Offline phase:** The parties execute Protocol 2 for $[\![\cdot]\!]_{(\tau,2\tau)}$ instead of $[\![\cdot]\!]$. They only check correctness for the $[\![\cdot]\!]$ part, and not for the $[\![\cdot]\!]_{(2t)}$ part. They obtain a random double sharing $([\![r]\!], [\![r]\!]_{(2t)})$.
- **Online phase:**
  1. The parties locally calculate $[\![\delta]\!]_{(2t)} := [\![x]\!] \cdot [\![y]\!] - [\![r]\!]_{(2t)}$.
  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 5. *Protocol 4 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 E.2. When evaluating a circuit gate-by-gate using Protocol 4, we consider an optimization in which we do not 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 [31]. However, this is not a problem in our case since the degree-2τ sharings have no redundancy in them. As shown in [31], 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.

Reducing communication using pseudo-randomness [10,37]. Our protocol as described so far is information-theoretic. We can reduce communication by using a pseudo-random generator in the following way. Assume that each pair of parties hold a joint random seed. Then, when party P_i shares an element with degree t, it is possible to derive t shares from the seed known to P_i and the corresponding party, and set the remaining $t+1$ shares (including the dealer's own share) given the pseudo-random shares and the value of the secret. Thus, only t shares need to be transmitted, thereby reducing communication by half. Using the same reasoning, it is possible to share a secret using $2t$-degree without any interaction. Here $n-1=2t$ shares are computed using the seed known to the dealer and each party, and then the dealer sets its own share such that all shares will reconstruct to the secret. We can use this idea to also reduce communication in the multiplication protocol. Instead of broadcasting δ, party P_i can share it to the parties with degree t, and use the above optimization, such that P_1 will have to send t elements instead of $n-1$. We note that here instead of comparing δ (to ensure correctness of output sharings), the parties can perform a batch correctness check (Protocol 3) for all sharings dealt by P_1 before the verification step in the main protocol.

Efficiency analysis. Assuming pairwise PRGs, the parties can generate $n-t \approx n/2$ random sharings $[\![r]\!]$ by communicating $n(n/2)$ elements in R, for an amortized communication complexity of n R-elements per random sharing. For $\mathcal{F}_{\mathsf{mult}}$, parties use one random double sharing $([\![r]\!], [\![r]\!]_{(2t)})$ and then communicate $3/2 \cdot n$ elements in $\mathbb{Z}_{2^{k+s}}$ in the online phase. Note that a random $2t$-sharing can be generated without interaction. The total communication cost of the passively secure instantiation is therefore $\mathcal{C}_{\mathsf{rand}}(k+s) = (k+s) \cdot n \log n$ and $\mathcal{C}_{\mathsf{mult}}(k+s) = (k+s) \cdot n \log n + (k+s) \cdot n \cdot 3/2$. For the compiled protocol the overall cost per multiplication gate in the circuit, expressed in bits, is

$$2\,\mathcal{C}_{\mathsf{mult}}(k+s) + \mathcal{C}_{\mathsf{rand}}(k+s) = (k+s)(3n \log n + 3n).$$

6 Implementation and Evaluation

We report in the following section on an implementation of both the Shamir based instantiation, as well as the 3-party instantiation based on replicated secret-sharing.

6.1 Implementation Details

We implement both protocols in C++ and rely on `uint64_t` and `unsigned int128`[8] types for arithmetic over $\mathbb{Z}_{2^\ell}$, where the former is used when $\ell = 64$

[8] This type is a GCC extension, cf. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html

and the latter when $\ell = 128$. Notice that this choice allows us to investigate two sets of parameters: $\ell = 64$ can be viewed as 32 bit computation with 32 bits of statistical security, while $\ell = 128$ gives us 64 bits of computation with 64 bits of statistical security. We rely on libsodium for hashing and the PRG we use is based on AES.

For the Galois-ring variant our implementation uses the 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 [16]). Concretely, a multiplication in $\mathsf{GR}(2^{64}, 4)$ require 20 `uint64_t` multiplications and 18 additions, while a multiplication in $\mathbb{Z}_{2^{64}}$ require only a couple of `uint64_t` multiplications as well as a few bitwise operations. so while some MPC primitives in $\mathbb{Z}_{2^\ell}$ may be cheaper (for example, masking a value in $\mathbb{Z}_{2^\ell}$ is cheaper), this gain in efficiency is greatly reduced by the complexity of operating in the Galois-Ring.

Experimental setup. We run our experiments on AWS `c5.9xlarge` machines, which have 36 virtual cores, 72gb of memory and a 10Gpbs network. We utilize 3 separate machines and so for experiments with $n > 3$, some parties run on the same machine. However, the load on each machine is distributed evenly (e.g., with 5 parties, the first two machines each run 2 parties each while the last run only 1 party).

6.2 Experiments

Our experiments comprises two points of comparison:

First we compare our Shamir based instantiation against the field protocol of [16]. For this, we use the implementation at [1]. We perform the same benchmarks as reported on in [16]; that is, circuits of varying depth with a fixed number of parties. Each experiment is repeated for n set to 3, 5, 7 and 9. The main goal here is to understand the overhead of working with $\mathsf{GR}(2^\ell, d)$ as opposed to working over $\mathbb{Z}_p$. As [1] supports different choices of the prime p we set p to be a 61-bit Mersenne prime, as this is the most efficient field that also allows for a reasonable expressive computations.

Our second set of experiments will compare our replicated instantiation against the protocols for computation over rings presented in [25]. In these experiments we measure the throughput of multiplications in our protocol; that is, how many multiplications our protocol can compute per second. Since we do not have access to the implementations of [25], we opt instead to use the experimental setup as theirs, in order to obtain a fair comparison. We report here on benchmarks run in a LAN setting.

While the protocol of [16] is the natural choice for comparing our n-party instantiation, a number of efficient specialized 3 party protocols exist which we briefly mention here. We choose the protocols of [25] for comparison as their experiments and setup is straightforward to replicate with our protocol, thus allowing us to make a fair comparison. Concurrently with [25], several other proposals for 3 party protocols have been published, such as [15] or [38]. However, no public implementation exist for these works, and the nature of the experiments they perform makes it very hard to perform a fair comparison (as we do later with the results from [25]). More precisely, both [15] and [38] evaluate their protocols relative to an implementation of ABY3 [35] that was also implemented by the authors themselves (as no public implementation of ABY3 was available at that time).

While [38] have better amortized communication cost, we estimate that their *conrete* running time (when considering end-to-end times, as we do in this work) will be worse. We base this conjecture on the fact that [38] uses the interpolation based check from [9]. For the case of fields, this check was shown in [11] to take several seconds in order to check 1 million multiplications (which is the benchmark we use). Running the same check, but over a ring, requires computation over a fairly large extension of $\mathbb{Z}_{2^k}$, which we have no reason to expect would be significantly faster than the field based check. Concluding, we would not be surprised if [38] is faster *in the online phase*; however, preprocessing the triples needed to get this would be much slower than our protocol. We stress that our protocol (for the 3 party case) has *no preprocessing*, so we expect our protocol to perform much better when measuring end-to-end times. We elaborate a bit more on the cost of the kind of check used in [38] later, when we discuss [11].

6.3 Results: Shamir instantiation

The results of our experiments can be seen in Table 1. Across the board, we see that preprocessing is more expensive in our protocol than in [16]. However, the overhead is in lines with the observation made above that operating in $\mathsf{GR}(2^\ell, d)$ is about 4 times as expensive than in $\mathbb{Z}_p$ when $\ell = 64$ and p is a 61-bit Mersenne prime. This is in particular true when the number of parties is small as here local computation is the dominant factor. Moving to a larger number of parties, the overhead decreases, which we attribute to differences in the efficiency of the communication layer between our protocol and the one in [16].

Interestingly, we see for a lower number of parties but for very deep circuits, that our protocol performs better in the online phase. E.g., [16] takes 7.3 seconds, while both of our version is below 4.5 seconds. One reason for this could again be differences in the communication layer (since both our protocols communicate roughly the same amount of information due to the fact that we only need to send a $\mathbb{Z}_{2^\ell}$ element during reconstruction). However, our protocol is again less efficient when the number of parties increase, which would be due to the fact that the king needs to send more data during reconstruction, as well as the increased cost of the broadcast when more parties are involved (one could distribute the role of the king among the parties, so that everyone handles an equal amount of reconstructions).

Finally, we see an expected overhead of roughly $\times 2$ between $\ell = 64$ and $\ell = 128$ (consider the depth 20 row in Table 1, as this is the setting where differences in local computation is most prominent.) This more or less confirms the intuition that an operation in $\mathbb{Z}_{2^{128}}$ is around 2-3 times as expensive compared to an operation in $\mathbb{Z}_{2^{64}}$.[9]

Depth	Protocol	3	5	7	9
20	Ours $\ell = 64$	1.56 / 0.18	2.12 / 0.28	2.46 / 0.37	2.70 / 0.47
	Ours $\ell = 128$	2.79 / 0.52	4.28 / 0.74	4.73 / 0.91	5.10 / 1.11
	[16]	0.43 / 0.18	0.63 / 0.22	0.93 / 0.45	1.03 / 0.28
100	Ours $\ell = 64$	1.50 / 0.23	1.97 / 0.30	2.30 / 0.37	2.76 / 0.41
	Ours $\ell = 128$	2.80 / 0.51	3.78 / 0.61	4.15 / 0.77	5.02 / 0.95
	[16]	0.42 / 0.42	0.64 / 0.22	0.90 / 0.52	1.04 / 1.27
$1,000$	Ours $\ell = 64$	1.58 / 0.67	1.95 / 1.08	2.23 / 1.43	2.62 / 1.84
	Ours $\ell = 128$	2.80 / 1.23	3.68 / 1.81	4.23 / 2.08	5.03 / 2.47
	[16]	0.41 / 0.96	0.63 / 0.68	0.89 / 0.95	1.05 / 1.17
$10,000$	Ours $\ell = 64$	1.50 / 3.85	2.01 / 8.55	2.41 / 13.41	2.65 / 16.76
	Ours $\ell = 128$	2.81 / 4.43	3.71 / 8.07	4.38 / 13.31	5.03 / 16.43
	[16]	0.38 / 7.30	0.61 / 7.32	0.89 / 8.40	1.05 / 12.88

Table 1: 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.

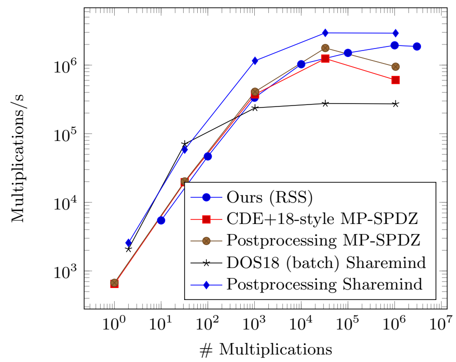
Comparing our instantiation with [16]. It is worth remarking that, for more elaborate protocols such as bit decompositions or truncations, operating over a prime field requires additional space for masking. For example, if we require 40 bits of security for masking, the 61-bit Mersenne prime only leaves room for ≈ 21 bits of computation. For these applications therefor, it is more reasonable to compare the numbers for [16] in Table 1 with our protocol with $\ell = 64$ (since $\mathbb{Z}_{2^k}$ does not require this extra space, $\ell = 64$ gives us 24 bits of computation at 40 bits of security). Alternatively, one could move to a 89-bit Mersenne or 127-bit Mersenne prime (allowing 49 and 87 bits of computation with 40 bits of security); however efficient multiplication in these fields require multiplication of essentially 128-bit integers without overflow, bringing it closer to operating in $GR(2^{128}, d)$.

6.4 Results: Replicated instantiation

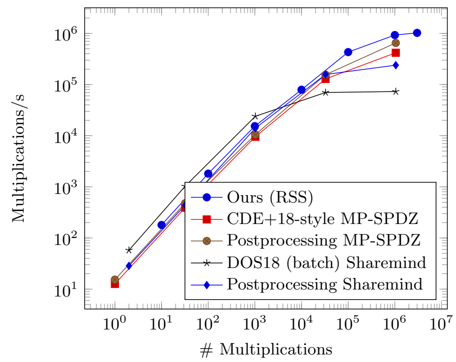
We also compare our replicated instantiation with the protocols of [25], results of which can be seen in Figure 1a and Figure 1b.[10] As we do not have access to

[9] Indeed, while a multiplication in $\mathbb{Z}_{2^{64}}$ is one unsigned 64-bit width multiplication, a multiplication on 128-bit wide types compile to three $\mathbb{Z}_{2^{64}}$ multiplications. That the overhead is less than $3x$ can be attributed to the compiler being able to easier vectorize 64-bit multiplications in the $\mathbb{Z}_{2^{128}}$ case.

[10] We thank the authors of [25] for giving us the tikz code of their graph.



(a) Throughput in LAN



(b) Throughput in WAN

Fig. 1: Throughput benchmarks for replicated secret-sharing with 3 parties.

the code of all the protocols considered in [25], we run our protocol in the same setup. With the exception of the Sharemind postprocessing protocol, we observe that we outperform all protocols of [25]. We may attribute this to the fact that both Sharemind and MP-SPDZ are more mature codebases and thus it is likely that a greater effort has been put into optimizations.

However, when we consider our protocol running in a WAN, we see that we outperform all protocols in [25]. This concurs with the fact that our protocol only needs to send 2 ring elements per multiplication, while the postprocessing protocols of [25] needs to send 3.

References

1. Fast large-scale honest-majority mpc for malicious adversaries, 2017. `https://github.com/cryptobiu/MPCHonestMajorityNoTriples`.
2. M. Abspoel, R. Cramer, I. Damgård, D. Escudero, and C. Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. Theory of Cryptography Conference TCC, 2019.
3. T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. pages 843–862, 2017.
4. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. pages 805–817, 2016.
5. Z. Beerliová-Trubíniová and M. Hirt. Efficient multi-party computation with dispute control. pages 305–328, 2006.
6. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. pages 663–680, 2012.
7. M. Blanton, A. Kang, and C. 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`.

8. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. pages 192–206, 2008.
9. D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, pages 67–97, 2019.
10. D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. pages 67–97, 2019.
11. E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 869–886, 2019.
12. R. Canetti. Security and composition of multiparty cryptographic protocols. 13(1):143–202, Jan. 2000.
13. I. Cascudo, R. Cramer, C. Xing, and C. Yuan. Amortized complexity of information-theoretically secure MPC revisited. pages 395–426, 2018.
14. O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. pages 182–199, 2010.
15. H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, pages 81–92, 2019.
16. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. pages 34–64, 2018.
17. R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. pages 769–798, 2018.
18. A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.
19. I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. pages 285–304, 2006.
20. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. pages 1–18, 2013.
21. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. pages 572–590, 2007.
22. I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. pages 799–829, 2018.
23. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. pages 643–662, 2012.
24. 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.
25. H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use your brain! arithmetic 3pc for any modulus with active security. In *1st Conference on Information-Theoretic Cryptography, ITC 2020, June 17-19, 2020, Boston, MA, USA*, pages 5:1–5:24, 2020.

26. J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. pages 225–255, 2017.
27. D. Genkin, Y. Ishai, and A. Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. pages 721–741, 2015.
28. D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. Circuits resilient to additive attacks with applications to secure computation. pages 495–504, 2014.
29. O. Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.
30. S. Goldwasser and Y. Lindell. Secure multi-party computation without agreement. 18(3):247–287, July 2005.
31. V. Goyal, Y. Liu, and Y. Song. Communication-efficient unconditional MPC with guaranteed output delivery. pages 85–114, 2019.
32. M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. pages 830–842, 2016.
33. Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. pages 259–276, 2017.
34. P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. pages 35–52, 2018.
35. P. Mohassel and P. Rindal. ABY3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. https://eprint.iacr.org/2018/403.
36. P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. pages 19–38, 2017.
37. P. S. Nordholt and M. Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. pages 321–339, 2018.
38. A. Patra and A. Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.
39. M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
40. S. Wagh, D. Gupta, and N. Chandran. SecureNN: Efficient and private neural network training. Cryptology ePrint Archive, Report 2018/442, 2018. https://eprint.iacr.org/2018/442.

Supplementary Material

A Secret Sharing

Definition 1. *A perfect* (t, 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* $x_i \in (\mathbb{Z}_{2^\ell})^m$ *is intended for party* P_i*.*[11]
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* $\mathsf{correct}$ *as defined below in Definition 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* $\mathsf{sharecons}$ *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* $[\![x]\!] + [\![y]\!]$*.*[12]

We also assume the following, non-standard properties:

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* $[\![x]\!]_\ell \to [\![x]\!]_{\ell'}$*.*

[11] 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 $\mathbb{Z}_{2^\ell}^{\log n}$.

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

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 $x' = x/2$.*

Next, 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 $\perp$).

Definition 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 $\mathsf{val}([\![v]\!])_J = v'$.*

B Some Missing Functionalities

Efficient Sum of Products. This the functionality presented in Section 2.3

Functionality 2 $\mathcal{F}_{\mathsf{DotProduct}}(\ell)$

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.

B.1 Input

This is the functionality used in Section 3.1

Functionality 3 $\mathcal{F}_{\text{input}}(\ell)$

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

C $\mathcal{F}_{\text{CheckZero}}$ – 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 5.1 and 5.2 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 4. $\mathcal{F}_{\text{CheckZero}}$ 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 4 $\mathcal{F}_{\text{CheckZero}}(\ell)$

The ideal functionality $\mathcal{F}_{\text{CheckZero}}$ receives $[\![v]\!]_\ell^H$ from the honest parties and uses them to compute v and $[\![v]\!]_\ell^{\mathcal{C}}$, the shares of v of the corrupt parties.

Then, $\mathcal{F}_{\text{CheckZero}}$ hands $[\![v]\!]_\ell^{\mathcal{C}}$ 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 [14,24]) 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 7. 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 [17]. 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 5.

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 [26], 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 5 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 5, 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 for the sacrifice step in [17].

Lemma 6. *If the check at the end of the first step in Protocol 5 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 5. 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 1. *Protocol 5 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 6, this event happens with probability of at most 2^{-s}, which is exactly the statsitcal error allowed by the proposition. ∎

$\mathcal{F}_{\textbf{randBit}}$ - 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 [24], 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 2. *Protocol 6 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 $[\![b]\!]_\ell^{\mathcal{C}}$ 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 [24], 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. ■

Protocol 6 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 $[\![d]\!]_{\ell+2} \rightarrow [\![d]\!]_{\ell+1}$, 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 7. 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 7 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 $[\![r_i]\!]_{\ell} \rightarrow [\![r_i]\!]_1$ 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. *Protocol 7 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 $[\![v]\!]_{\ell}^{\mathcal{C}}$ 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 $[\![r_0]\!]_{\ell}^{\mathcal{C}}, \ldots, [\![r_{\ell-1}]\!]_{\ell}^{\mathcal{C}}$ from $\mathcal{A}$.
2. $\mathcal{S}$ locally computes $[\![r]\!]_{\ell}^{\mathcal{C}} = \sum_{i=0}^{\ell-1} [\![r_i]\!]_{\ell}^{\mathcal{C}}$ and $[\![c]\!]_{\ell}^{\mathcal{C}} = [\![v]\!]_{\ell}^{\mathcal{C}} + [\![r]\!]_{\ell}^{\mathcal{C}}$. Then, it chooses a random $c \in \mathbb{Z}_{2^\ell}$ and computes $[\![c]\!]_{\ell}^{H} = \mathsf{share}(c, [\![c]\!]_{\ell}^{\mathcal{C}})$.
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 $[\![r_i]\!]_\ell^{\mathcal{C}} \to [\![r_i]\!]_1^{\mathcal{C}}$ 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 $[\![T]\!]_1^{\mathcal{C}}$.
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 $\mathsf{share}(b, [\![T]\!]_1^{\mathcal{C}})$ to obtain the honest parties' shares $[\![b]\!]_1^{H}$.
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 $c = v + r \bmod 2^\ell$ 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. ∎

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 7 (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.

D 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 [33,34], 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 [34,4], which is described in 8. 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 8 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 [33]. We note that this can be extended to instantiate $\mathcal{F}_{\mathsf{DotProduct}}$ at the communication cost of one single multiplication, as shown in [16].

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 9 in the $\mathcal{F}_{\mathrm{RO}}$-hybrid model. The $\mathcal{F}_{\mathrm{RO}}$ functionality is described in Functionality 5.

We remark that that this protocol does not instantiate $\mathcal{F}_{\mathrm{CheckZero}}$ 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 $\mathcal{F}_{\mathrm{CheckZero}}{}'$. This is minor change since the main requirement from $\mathcal{F}_{\mathrm{CheckZero}}$ 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 5 $\mathcal{F}_{\mathrm{RO}}$ – 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 9 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 4. *Protocol 9 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.

■

E Proofs for Section 5.2 - Shamir-SS Instantiation

E.1 Proof of Lemma 2: Securely Computing $\mathcal{F}_{\mathsf{rand}}$

Lemma 7 (Lemma 2 - restated). *Protocol 2 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{rand}}$ 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 3, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{rand}}$.
2. If the checks pass but the honest parties' shares are inconsistent, $\mathcal{S}$ outputs fail. By Lemma 3 this only happens with probability at most $2^{-\kappa}$, allowed by the claim.

3. In the remaining case, the checks of Protocol 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 $\mathcal{F}_{\mathsf{rand}}$.

Before the invocation of $\mathcal{F}_{\mathsf{rand}}$, 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. ■

E.2 Proof of Lemma 5: Securely Computing $\mathcal{F}_{\mathsf{mult}}$

Lemma 8 (Lemma 5 - restated). *Protocol 4 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 2. By the proof, we have that $[\![r]\!]$ is a correct sharing. The sharing $[\![r']\!]_{(2t)}$ 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 $[\![\delta]\!]_{(2t)} \leftarrow [\![\cdot]\!]_{(2t)}$. 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.[13] In the real execution, the shares also correspond to z. The sharing in the real execution is calculated as $[\![r]\!] + \delta$, where $[\![r]\!]$ is a uniformly random sharing. Therefore, the outputs are identical in both executions. ■

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