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

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

[1] CWI, Amsterdam, The Netherlands
abspoel@cwi.nl
[2] Partisia, Aarhus, Denmark
[3] Aarhus University, Aarhus, Denmark
escudero@cs.au.dk
[4] Technion, Haifa, Israel
ariel.nof@cs.technion.ac.il

Abstract. Multiparty computation (MPC) over rings such as $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ has received a great deal of attention recently due to its ease of implementation and attractive 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 novel compiler for MPC over the ring $\mathbb{Z}_{2^k}$ in the honest majority setting that turns a semi-honest protocol into an actively secure protocol with very little overhead. The communication cost per multiplication is only twice that of the semi-honest protocol, making the resultant actively secure protocol almost as fast.

To demonstrate the efficiency of our compiler, we implement both an optimized 3-party variant (based on replicated secret-sharing), as well as a protocol for n parties (based on a recent protocol from TCC 2019). For the 3-party variant, we obtain a protocol which outperforms the previous state of the art that we can experimentally compare against. Our n-party variant is the first implementation for this particular setting, and we show that it performs comparably to the current state of the art over fields.

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

A. Dalskov—Work done while author was a student at Aarhus University.

K. Sako and N. O. Tippenhauer (Eds.): ACNS 2021, LNCS 12727, pp. 122–152, 2021.
https://doi.org/10.1007/978-3-030-78375-4_6

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) [5,6,14,18,22,29,31].

1.1 Secure Computation over Rings

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 [15], while several other works focus on honest majority case for small number of parties [2,14,21,22]. Computation over $\mathbb{Z}_{2^k}$ is appealing as it is generally more natural than computation modulo a prime, especially for powers like 2^{32}, 2^{64} or 2^{128}. This type of computation has the potential to lead to more efficient protocols with respect to computation over fields, as in practical settings it avoids a software implementation of a modular reduction operation by using native data-types existing in modern architectures. For example, computing fast reductions modulo an n-bit Mersenne prime requires computing a product of two n-bit numbers without overflow.[1] Thus, for a ≈128-bit prime, this requires arithmetic on 256-bit numbers. In contrast, arithmetic in $\mathbb{Z}_{2^{128}}$ is supported by most modern compilers. Furthermore, many MPC applications require bitwise operations, like secure comparison to be able to perform branching, or secure truncation to be able to handle fixed-point data. This is particularly relevant for machine learning applications, for example. Protocols based on computation modulo 2^k have the potential to execute these operations much more efficiently, given the existing compatibility between binary computation, that is, computation modulo 2, and operations modulo a larger power of 2.

The improvement in performance of ring-based protocols was observed experimentally for the aforementioned SPD$\mathbb{Z}_{2^k}$ protocol in [17]. More recently, the work by Dalskov et al. [16] demonstrated that the same applies for honest majority protocols, where the protocols over rings presented in that work outperform similar ones over fields by a factor of around 2.

1.2 Our Contributions

As discussed above, it is a natural and well-motivated question to study the efficiency of MPC protocols over $\mathbb{Z}_{2^k}$. In spite of the benefits that this algebraic structure may provide, protocol design becomes much harder due to the undesired properties of this ring, like the existence of zero-divisors. For example, to date, no concretely-efficient protocol over $\mathbb{Z}_{2^k}$ that works for any number of parties has been proposed in the honest majority setting. This is particular critical when active adversaries are considered, as techniques to ensure security in this

[1] This reduction uses the identity $x \cdot y = a2^n + b \equiv a + b \bmod 2^n - 1$ for some a, b. However this requires computing and storing the product $x \cdot y$ without overflow.

case typically rely on properties of fields. In this work, we push the knowledge barrier on this area by presenting a generic compiler that transforms a passively secure protocol for computation over $\mathbb{Z}_{2^{k+s}}$ in the honest majority setting, to a protocol over the ring $\mathbb{Z}_{2^k}$ that is actively secure with abort and provides roughly s bits of statistical security. Summarizing our contributions:

- Our compiler simplifies protocol design by only requiring that the underlying passively secure protocol is secure up to an additive attack, which is a condition that is much easier to ensure. For example, this was shown to hold for multiple well-known protocols over fields in [24], a result which we extend in our paper to recent protocols over rings.
- Our compiler is highly efficient and the overhead is essentially just twice that of the passively secure protocol. More precisely, each multiplication just needs to be evaluated twice.
- Our compiler preserves all the properties of the passively secure protocol. In particular, we obtain the first actively secure protocols where the cost of dot products is *independent* of their length without relying on expensive function dependent preprocessing such as is the case for prior work [13,21,22,34].
- Finally, we provide two instantiations and show through experiments that they are concretely efficient:
  1. Our first instantiation is for 3 parties and is based on replicated secret sharing. We show experimentally as well as theoretically that it outperforms other 3 party protocols both over the ring $\mathbb{Z}_{2^{k+s}}$ and over fields $\mathbb{Z}_p$ with $\log(p) \approx k + s$. This gap of s bits for the field case is necessary when considering applications that require more complex primitives like secure comparison or truncation, as traditional techniques for these tasks (e.g. [12]) require such a gap to guarantee privacy.
  2. Our second instantiation is for an arbitrary number n of parties, and is based on the work by Abspoel et al. [2]. It is the first *practical* (in the sense of having been experimentally demonstrated to be concretely efficient) example of such a protocol for $\mathbb{Z}_{2^k}$ with active security and an honest majority. The protocol from [2] requires $3(k + s)\log n$ bits per multiplication in the online phase; however we describe a novel optimization that removes the $\log n$ factor that might be of independent interest. Although our protocol does not outperform its field counterpart from [14] (it is merely comparable), our results illustrate that the *a priori* benefits of working over $\mathbb{Z}_{2^k}$ may be outweighed by the complexity of computing over the so-called Galois ring extensions, which are required to make these protocols work. This observation is relevant as many recent works, such as [8,9,34], rely on Galois ring extensions of large degree without taking into account their computational overhead.

Outline. Section 2 introduces some of the definitions we will be needing and Sect. 3 introduces the building blocks we need in our compiler. In Sect. 4 our main protocol (i.e., our compiler) is presented, as well as the formal statements of security and security proofs. We then present the n party instantiation in

Sect. 5, and a three party instantiation in Sect. 6. Finally, in Sect. 7 we present our experimental results and compare our results with prior works.

1.3 Related Work

The only previous general compiler with concrete efficiency over rings, to the best of our knowledge, is the compiler of [20], which was improved by [21]. However, their compiler does not preserve the adversary threshold when moving from passive to active security. In addition, in [20] and [21] 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 [15] which was proven to be practical in [17]. 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.

The work of [30] provides a method for working over small fields (e.g., $\mathbb{F}_2$) which improves upon the Chida et al. protocol [14]. However, their method is not suited for the rings that we consider in our work.

In the three-party setting with one corruption, there are several works which provide high efficiency for arithmetic computations over rings. The Sharemind protocol [7] is being used to solve real-world problems but provides only passive security. The actively secure protocol of [22], which was optimized and implemented in [3], is based on the "cut–and–choose" approach and will be favorable when working over small rings. The actively secure three-party protocol of [21] 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 [21] in Sect. 7.3. Finally, a new promising direction was presented by [9], but their verification step takes several seconds for a 1-million gate over fields, and this is expected to be orders of magnitude worse for rings due to the need of large-degree Galois ring extensions. The protocols of [13, 34] have a slightly higher bandwidth overall than [3], but they focus on minimizing online (input-dependent) cost and they tailor their protocols to specific applications for machine learning. Also, [34] uses the techniques from [9] for the preprocessing, so it is unlikely to provide any efficiency in practice.

Finally, it is important to mention that the techniques from [9], which work for 3 parties, can be generalized to multiple parties as a passive-to-active compiler. This has been done in [28] over fields, and it is not hard to see that these techniques can be made to work over $\mathbb{Z}_{2^k}$ by considering large-degree Galois ring extensions, as done in [9]. However, this method is not practical as even a small degree extension can be quite expensive, as shown in this work. Furthermore, the round complexity of the passively secure protocol is not preserved by this transformation.

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, hence $t < \frac{n}{2}$. 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 $x \equiv_\ell y$.

We use the standard definition of security based on the ideal/real model paradigm [10,25], with security formalized for non-unanimous abort. This means that the adversary first receives the output, and then determines for each honest party whether they will receive abort or receive their correct output. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution [26]. 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.

2.1 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 Sect. 2.2. The only extra, non-standard properties required by our compiler are the following (for a formalization of the requirements of the SSS, see the full version of this work):

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. 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 $\perp$).

2.2 Secure Multiplication up to Additive Attacks [23,24]

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 protocol 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 the functionality $\mathcal{F}_{\mathsf{mult}}$, which receives input sharings $[\![x]\!]$ and $[\![y]\!]$ from the honest parties, and an additive error 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 secret-shared value.

The requirements defined by this functionality can be met by several semi-honest multiplication protocols over $\mathbb{Z}_{2^\ell}$. In this work we focus on two of them in particular: one based on replicated secret sharing, and the other a more recent protocol of Abspoel et al. [2], which extends Shamir's secret sharing to the setting of $\mathbb{Z}_{2^\ell}$.

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. As in [14], we will show that the functionality can be realized at almost the same cost as $\mathcal{F}_{\mathsf{mult}}$.

[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.

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, a core step of our compiler is checking that a secret-shared value is zero, leaking nothing more than this binary information. This is not easy to instantiate over $\mathbb{Z}_{2^k}$, and we discuss this in Sect. 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 Sect. 2.1.

$\mathcal{F}_{\mathsf{rand}}$ – **Generating Random Coins.** 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, and we discuss concrete instantiations later on.

$\mathcal{F}_{\mathsf{coin}}$ – **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.

$\mathcal{F}_{\mathrm{bc}}$ – **Broadcast with Abort.** With this functionality, 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-upon value is precisely the one that the sender sent. The protocol may abort, and can be instantiated using the well-known echo-broadcast protocol, where the parties echo the message they received and send it the other parties.

$\mathcal{F}_{\mathrm{input}}$ – **Secure Sharing of Inputs.** This is a functionality that allows a party to distribute consistent shares of its input. This can be instantiated generically by sampling $[\![r]\!]$ using $\mathcal{F}_{\mathsf{rand}}$, reconstructing this value to the party who will provide input x, and letting this party broadcast the difference $x - r$. The parties can then compute the shares $[\![x]\!] = (x - r) + [\![r]\!]$.

3.1 Checking Equality to 0

For our compiler we require a functionality $\mathcal{F}_{\mathrm{CheckZero}}(\ell)$, 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.

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 unfortunately more expensive and complicated than the protocol over fields described above. On the upside, 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, for 3 parties for example, one can get a much more efficient solution, as we show in Sect. 6.

Our general protocol to compute $\mathcal{F}_{\text{CheckZero}}$ is described in Protocol 1. We consider two functionalities, $\mathcal{F}_{\text{CorrectMult}}$ and $\mathcal{F}_{\text{randBit}}$, that compute correct multiplications and sample shared random bits, respectively.

We have the following proposition.

Proposition 1. *Protocol 1 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.*

Protocol 1 Checking Equality to 0

Input: The parties hold a sharing $[\![v]\!]_\ell$.

The protocol:

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

Correct Multiplication. We consider a functionality $\mathcal{F}_{\text{CorrectMult}}$, that is similar to $\mathcal{F}_{\mathsf{mult}}$, except it does not allow additive errors. Our protocol to instantiate this functionality 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 "SPD$\mathbb{Z}_{2^k}$ trick" from [15]. 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 2.

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 [22], 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.

It can be easily checked that w in the protocol equals $d' - r \cdot d$, where d' and d are the additive errors from the two calls to $\mathcal{F}_{\mathsf{mult}}$. The following lemma shows that d cannot be non-zero with non-negligible probability, which shows that the triple $(⟦a⟧_{\ell+s}, ⟦b⟧_{\ell+s}, ⟦c⟧_{\ell+s})$ is correct modulo 2^k. From this, the security of Protocol 2 follows.

Lemma 1 **([15]).** *If the check at the end of the first step in Protocol 2 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 $1 - 2^{-s}$.*

Protocol 2 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 $⟦\pi⟧_{\ell+s} \rightarrow ⟦\pi⟧_\ell$ 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 $⟦z⟧_\ell$.

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. ■

Generating Random Shared Bits. We also consider a functionality $\mathcal{F}_{\text{randBit}}$ that operates in a similar way to $\mathcal{F}_{\mathsf{rand}}$, but ensures the random shared value is in $\{0, 1\}$. We instantiate this functionality essentially by showing that the bit-generation procedure from [17], 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 Sect. 2.1, which states that parties can locally divide their shares of a secret x mod 2^ℓ by 2 to obtain shares of $x/2$ mod $2^{\ell-1}$, as long as the shares and the secret are even.

Proposition 2. *Protocol 3 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 $\llbracket b \rrbracket_\ell^{\mathcal{C}}$ 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' output 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 [17], 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 $\llbracket d \rrbracket_{\ell+2} + 1 = 2 \cdot \sqrt{c}^{-1} \llbracket r \rrbracket_{\ell+2} + (\sqrt{c}^{-1} + 1)$. Now, the shares $2 \cdot \sqrt{c}^{-1} \llbracket r \rrbracket_{\ell+2}$ 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 Sect. 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 3 Random Shared Bits Generation

The protocol:

1. The parties call $\mathcal{F}_{\mathsf{rand}}(\ell + 2)$ to obtain a shared value $\llbracket r \rrbracket_{\ell+2}$. Then, the parties set $\llbracket a \rrbracket_{\ell+2} = 2 \cdot \llbracket r \rrbracket_{\ell+2} + 1$.
2. The parties call $\mathcal{F}_{\text{CorrectMult}}(\ell + 2)$ on input $\llbracket a \rrbracket_{\ell+2}$ and $\llbracket a \rrbracket_{\ell+2}$ to obtain shares $\llbracket c \rrbracket_{\ell+2} = \llbracket a^2 \rrbracket_{\ell+2}$. Then, they run $\mathsf{open}(\llbracket c \rrbracket_{\ell+2})$ to obtain c.
3. The parties compute $\llbracket d \rrbracket_{\ell+2} = \sqrt{c}^{-1} \cdot \llbracket a \rrbracket_{\ell+2}$, 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 $\llbracket d \rrbracket_{\ell+2} \to \llbracket d \rrbracket_{\ell+1}$, and compute $\llbracket b' \rrbracket_{\ell+1} = \llbracket d \rrbracket_{\ell+1} + 1$.
5. The parties locally shift their shares of b' one position to the right to obtain shares $\llbracket b \rrbracket_\ell$, where $b \equiv_\ell \frac{b'}{2}$.

Outputs: The parties output $\llbracket b \rrbracket_\ell$.

4 The Main Protocol for Rings

In this section, we present our construction to compute arithmetic circuits over the ring $\mathbb{Z}_{2^k}$. A formal description appears in Protocol 4. Our protocol follows the paradigm of [14], which roughly works by running a "redundant" copy of the circuit where each shared wire value $[\![w]\!]$ is accompanied by $[\![r \cdot w]\!]$ for some global uniformly random r. In [14] it was shown that such a "dual" execution allows the parties to perform a simple check to ensure that no additive errors were introduced in the multiplication gates. However, such check does not directly work over $\mathbb{Z}_{2^k}$, given that it relies on the fact that every non-zero element must be invertible, which only holds over fields.

In order to reduce the cheating success probability, we borrow the idea of [15] of working on the larger ring $\mathbb{Z}_{2^{k+s}}$. As we will show below, this ensures that a similar check to that in [14] over fields can be carried out over $\mathbb{Z}_{2^{k+s}}$, ensuring no additive attacks over $\mathbb{Z}_{2^k}$ are carried out, except with probability at most 2^{-s}.

At the core of the security of our protocol lies the following lemma, which shows that an additive attack that is non-zero modulo 2^k in any multiplication gate leads to failure in the final check to zero, with overwhelming probability.

Protocol 4 Computing Arithmetic Circuits Over the Ring $\mathbb{Z}_{2^k}$

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

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

The protocol:

1. *Secret sharing the inputs:*
   (a) For each input x_j held by party P_j, party P_j represent it as an element of $\mathbb{Z}_{2^{k+s}}^L$ and sends x_j to $\mathcal{F}_{\text{input}}(k+s)$.
   (b) Each party P_j records its vector of shares $(x_1^j, \ldots, x_M^j)$ of all inputs, as received from $\mathcal{F}_{\text{input}}(k+s)$. If a party received $\perp$ from $\mathcal{F}_{\text{input}}$, then it sends abort to the other parties and halts.
2. *Generate randomizing shares:* The parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![r]\!]_{k+s}$, where $r \in_R \mathbb{Z}_{2^{k+s}}$.
3. *Randomization of inputs:* For each input wire sharing $[\![v_m]\!]_{k+s}$ (where $m \in \{1, \ldots, M\}$) the parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![r]\!]_{k+s}$ to receive $[\![r \cdot v_m]\!]_{k+s}$.
4. *Circuit emulation:* The parties traverse over the circuit in topological order. For each gate G_ℓ the parties work as follows:
   - *G_ℓ is an addition gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively, the parties locally compute $([\![x+y]\!]_{k+s}, [\![r \cdot (x+y)]\!]_{k+s})$.
   - *G_ℓ is a multiplication-by-a-constant gate:* Given a constant $a \in \mathbb{Z}_{2^k}$ and tuple $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ on the input wire, the parties locally compute $([\![a \cdot x]\!]_{k+s}, [\![r \cdot (a \cdot x)]\!]_{k+s})$.
   - *G_ℓ is a multiplication gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively:
     (a) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![x \cdot y]\!]_{k+s}$.
     (b) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[\![r \cdot x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![r \cdot x \cdot y]\!]_{k+s}$.
5. *Verification stage:* Let $\{([\![z_i]\!]_{k+s}, [\![r \cdot z_i]\!]_{k+s})\}_{i=1}^N$ be the tuples on the output wires of all multiplication gates and let $\{[\![v_m]\!]_{k+s}, [\![r \cdot v_m]\!]_{k+s}\}_{m=1}^M$ be the tuples on the input wires of the circuit.
   (a) For $m = 1, \ldots, M$, the parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![\beta_m]\!]_{k+s}$.
   (b) For $i = 1, \ldots, N$, the parties call $\mathcal{F}_{\mathsf{rand}}(k+s)$ to receive $[\![\alpha_i]\!]_{k+s}$.
   (c) *Compute linear combinations:*

i. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $(\llbracket \alpha_1 \rrbracket_{k+s}, \ldots, \llbracket \alpha_N \rrbracket_{k+s}, \llbracket \beta_1 \rrbracket_{k+s}, \ldots, \llbracket \beta_M \rrbracket_{k+s})$ and $(\llbracket r \cdot z_1 \rrbracket_{k+s}, \ldots, \llbracket r, \cdot z_N \rrbracket_{k+s}, \llbracket r \cdot v_1 \rrbracket_{k+s}, \ldots, \llbracket r \cdot v_M \rrbracket_{k+s})$ to obtain $\llbracket u \rrbracket_{k+s} = \llbracket \sum_{i=1}^{N} \alpha_i \cdot (r \cdot z_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m) \rrbracket_{k+s}$.

ii. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $(\llbracket \alpha_1 \rrbracket, \ldots, \llbracket \alpha_N \rrbracket, \llbracket \beta_1 \rrbracket, \ldots, \llbracket \beta_M \rrbracket)$ and $(\llbracket z_1 \rrbracket_{k+s}, \ldots, \llbracket z_N \rrbracket_{k+s}, \llbracket v_1 \rrbracket_{k+s}, \ldots, \llbracket v_M \rrbracket_{k+s})$ to obtain $\llbracket w \rrbracket_{k+s} = \llbracket \sum_{i=1}^{N} \alpha_i \cdot z_i + \sum_{m=1}^{M} \beta_m \cdot v_m \rrbracket_{k+s}$.

(d) The parties run $\mathsf{open}(\llbracket r \rrbracket_{k+s})$ to receive r.

(e) Each party locally computes $\llbracket T \rrbracket_{k+s} = \llbracket u \rrbracket_{k+s} - r \cdot \llbracket w \rrbracket_{k+s}$.

(f) The parties call $\mathcal{F}_{\mathrm{CheckZero}}(k+s)$ on $\llbracket T \rrbracket_{k+s}$. 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 $\llbracket v \rrbracket_{k+s}$, the parties locally convert to $\llbracket v \rrbracket_k$. Then, they run $v \bmod 2^k = \mathsf{open}(\llbracket v \rrbracket_k, j)$, 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.

Lemma 2. *If $\mathcal{A}$ sends an additive value $d \not\equiv_k 0$ in any of the calls to $\mathcal{F}_{\mathsf{mult}}$ in the execution of Protocol 4, then the value T computed in the verification stage of Step 5 equals 0 with probability bounded by $2^{-s+\log(s+1)}$.*

Proof: Suppose that $(\llbracket x_i \rrbracket_{k+s}, \llbracket y_i \rrbracket_{k+s}, \llbracket z_i \rrbracket_{k+s})$ is the multiplication triple corresponding to the i-th multiplication gate, where $\llbracket x_i \rrbracket_{k+s}, \llbracket y_i \rrbracket_{k+s}$ are the sharings on the input wires and $\llbracket z_i \rrbracket_{k+s}$ 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 $\llbracket r \cdot z_i \rrbracket_{k+s}$ 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 $\llbracket x_i \rrbracket_{k+s}$ and $\llbracket y_i \rrbracket_{k+s}$, 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 $\llbracket y_i \rrbracket_{k+s}$ and $\llbracket r \cdot x_i \rrbracket_{k+s}$. However, it is possible that the adversary has attacked previous gates and so $\llbracket y_i \rrbracket_{k+s}$ is actually multiplied with $\llbracket r \cdot x_i + \epsilon_i \rrbracket$, where the value $\epsilon_i \in \mathbb{Z}_{2^{k+s}}$ is an accumulated error from previous gates.[3] Thus, it holds that $\mathsf{val}(\llbracket z_i \rrbracket)^H = x_i \cdot y_i + \delta_i$ and $\mathsf{val}(\llbracket r \cdot z_i \rrbracket)^H = (r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i$. Similarly, for each input wire with sharing $\llbracket v_m \rrbracket$, it holds that $\mathsf{val}(\llbracket r \cdot v_m \rrbracket)^H = r \cdot v_m + \xi_m$, where $\xi_m \in \mathbb{Z}_{2^{k+s}}$ is the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with $\llbracket r \rrbracket_{k+s}$ and the shared input $\llbracket v_m \rrbracket_{k+s}$. Thus, we have that

[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 $\llbracket x_i \rrbracket_{k+s}$ at the current value on the wire, and then given the randomized sharing $\llbracket x_{i'} \rrbracket_{k+s}$, define $\epsilon_i = x_{i'} - r \cdot x_i$ as the accumulated error on the input wire.

$$\begin{aligned}\mathsf{val}([\![u]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) \\ &\qquad + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \varTheta_1 \\ \mathsf{val}([\![w]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \varTheta_2\end{aligned}$$

where $\varTheta_1 \in \mathbb{Z}_{2^{k+s}}$ and $\varTheta_2 \in \mathbb{Z}_{2^{k+s}}$ are the values being added by the adversary when $\mathcal{F}_{\mathsf{DotProduct}}$ is called in the verification step, and so

$$\begin{aligned}\mathsf{val}([\![T]\!])^H &= \mathsf{val}([\![u]\!])^H - r \cdot \mathsf{val}([\![w]\!])^H = \\ &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \theta_1 \\ &\qquad - r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \varTheta_2 \right) \\ &= \sum_{i=1}^{N} \alpha_i \cdot (\epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i) \qquad (1) \\ &\qquad + \sum_{m=1}^{M} \beta_m \cdot \xi_m + (\varTheta_{1-r} \cdot \varTheta_2),\end{aligned}$$

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 $\varDelta_i = \epsilon_i \cdot y_i + \gamma_{i-r} \cdot \delta_i$.

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

$$\beta_{m_0} \cdot \xi_{m_0} \equiv_{k+s} \left(- \sum_{i=1}^{N} \alpha_i \cdot \varDelta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\varTheta_{1-r} \cdot \varTheta_2) \right).$$

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

$$\beta_{m_0} \equiv_{k+s-u} \left(\frac{- \sum_{i=1}^{N} \alpha_i \cdot \varDelta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\varTheta_{1-r} \cdot \varTheta_2)}{2^u} \right) \cdot \left(\frac{\xi_{m_0}}{2^u} \right)^{-1}.$$

  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:

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

To sum up the proof, in the first case we obtained that $T = 0$ with probability of at most 2^{-s} whereas in the second case, this holds with probability of at most $2^{-s+\log(s+1)}$. Therefore, we conclude that the probability that $T = 0$ in the verification step is bounded by $2^{-s+\log(s+1)}$ as stated in the lemma. This concludes the proof. ■

The security of Protocol 4 now follows as Lemma 2 shows that additive errors that are non-zero modulo 2^k cannot take place without leading to abort. However, one non-trivial issue lies in handling additive attacks that are zero modulo 2^k, but not modulo 2^{k+s}, as these do not affect correctness but may lead to selective failure attacks, in which an abort signal can be generated depending on the inputs from honest parties. Our protocol deals with this potential attack by using secret coefficients for the random linear combination taken in the verification step. If we take public coefficients, as done in [14], the following attack can be carried out.

Assume 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 $T = \alpha_{i_0 s} \cdot d_{i_0} + \alpha_{j_0} \cdot (d_{i_0} \cdot w_{j_0}) = d_{i_0}(\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0})$. 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 output 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 $\sum_{i=1}^{N} [\![\alpha_i]\!] \cdot [\![z_i]\!]$.

We state the security of our protocol below. A full simulation-based proof appears in the full version of this work.

Theorem 1. *Let f be an n-party functionality over $\mathbb{Z}_{2^k}$ and let s be a statistical security parameter. Then, Protocol 4 securely computes f with abort in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\text{CheckZero}}, \mathcal{F}_{\mathsf{DotProduct}})$-hybrid model with statistical error $2^{-s+\log(s+1)}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties. The communication complexity in bits of the resulting protocol is*

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

where M is the number of inputs, N is the number of multiplication gates in the circuit, O is the number of output wires of the circuit and $\mathcal{C}_*$ represents the cost (in bits) of calling the functionality $\mathcal{F}_*$.

5 Instantiation for n Parties

In this section, we present our instantiation based on Shamir's secret sharing over rings, using the techniques from [2]. This technique works for any number of parties, although for 3 parties one can obtain more efficient solutions, such as the one we describe in Sect. 6 that uses replicated secret sharing. 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 $\alpha_0, \dots, \alpha_n \in R$ 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 the privacy parameter of the scheme. Then, the set of *correct* share vectors is

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

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 Sect. 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 $[\![x]\!]_\tau = (x_1, \dots, x_n)$. 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 $[\![x]\!]_\tau, [\![y]\!]_\tau$ sends $(x_1 y_1, \dots, x_n y_n) \mapsto x \cdot y$.

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

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

5.1 Generating Randomness

We efficiently realize $\mathcal{F}_{\mathsf{rand}}$ by letting each player P_i sample and secret-share a random element s_i, and then multiplying the resulting vector of n random elements with a particular[5] Vandermonde matrix [19].[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 6 by opening a random linear combination of the sharings and verifying the result.

Since our secret-sharing scheme $[\![\cdot]\!]_\tau$ 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 $[\![\cdot]\!]_\tau$ be interpreted as an R-linear secret-sharing scheme [2,11]. 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 $[\![\cdot]\!]_\tau$), 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 $([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$. 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: $a = a_0 + a_1X + \cdots + a_{d-1}X^{d-1}$. 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 $M_r([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$. If a single party P has a vector of shares $(s_1, \ldots, s_d) \in R$ for $\langle x\rangle = ([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$, then $M_r(s_1, \ldots, s_d)^T$ is their vector of shares corresponding to $\langle rx\rangle$.

[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.

In our protocol, the parties compute $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T = A(\langle s_1 \rangle, \ldots, \langle s_n \rangle)^T$, 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 $\langle s_k \rangle = ([\![s_{k1}]\!]_\tau, [\![s_{kd}]\!]_\tau)^\mathsf{T}$. 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 super-invertible, i.e. any square submatrix obtained by sampling a subset of $n-\tau$ columns is invertible [2].

Protocol 5 Generating random sharings of $[\![\cdot]\!]_\tau$

The protocol:

1. Each party P_i samples an element $s_i \leftarrow (\mathbb{Z}_{2^\ell})^d$ and secret-shares it as $\langle s_i \rangle$ among all parties.
2. The parties locally compute the linear matrix-vector product to obtain $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T := A(\langle s_1 \rangle, \ldots, \langle s_n \rangle)^T$.
3. The parties execute Protocol 6 $\lceil \kappa/d \rceil$ 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 $\langle r_j \rangle = ([\![r_{j1}]\!]_\tau, \ldots, [\![r_{jd}]\!]_\tau)$ and output $[\![r_{11}]\!]_\tau, \ldots, [\![r_{1d}]\!]_\tau, [\![r_{21}]\!]_\tau, \ldots, [\![r_{(n-\tau)d}]\!]_\tau$.

Lemma 3. *Protocol 5 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{rand}}$ for $[\![\cdot]\!]_\tau$ with statistical error of at most $2^{-\kappa}$ in the presence of a malicious adversary controlling $t < n/2$ parties.*

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

We distinguish three cases:

1. If at least one of the simulated honest parties aborts in any of the executions of Protocol 6, 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 4 this only happens with probability at most $2^{-\kappa}$, allowed by the claim.
3. In the remaining case, the checks of Protocol 6 pass and the honest parties' shares are consistent. $\mathcal{S}$ calculates the corrupted parties' shares $\langle r_1 \rangle_C, \ldots, \langle r_{n-\tau} \rangle_C$ 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 $\langle \cdot \rangle$.

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

Let $H \subseteq \mathcal{H}$ be a subset of honest parties of size $n - \tau$, and let $C := \{1, \ldots, n\} \backslash H$ 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 super-invertible), 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. ■

5.2 Checking Correctness of Sharings

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

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

Protocol 6 Checking correctness of sharings of $\langle \cdot \rangle$

Inputs: 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.

The 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 4. *If at least one of the input sharings $\langle x_1 \rangle, \ldots, \langle x_N \rangle$ is incorrect, Protocol 6 outputs* $\mathsf{correct}$ *with probability at most $\frac{1}{2^d}$.*

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

Lemma 5. *Let $C \subseteq R^n$ be a free R-module. Then for all $x \notin C$ and $u \in R^n$, we have that*

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

where r is chosen uniformly at random from R.

Proof: [Proof of Lemma 4]. 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 5 this probability is bounded by 1/2, since a_1 was chosen uniformly at random. Since $\langle u \rangle$ is masked with $\langle r \rangle$, the protocol is private. ■

5.3 Secure Multiplication up to Additive Attacks

Multiplication follows the outline of the passively secure protocol of [19]. The protocol begins with a preprocessing phase, where *random double sharings* are produced, i.e. a pair of sharings $(\llbracket r \rrbracket_\tau, \llbracket r \rrbracket_{(2\tau)})$ of the same uniformly random element r shared using polynomials of degree τ and degree 2τ, respectively.

We denote a double sharing as $\llbracket r \rrbracket_{(\tau,2\tau)} := ((r_1, r'_1), \ldots, (r_n, r'_n))$. It is a $\mathbb{Z}_{2^\ell}$-linear secret-sharing scheme with secret space $\mathbb{Z}_{2^\ell}$ and share space $R \oplus R$. The set of correct share vectors is the $\mathbb{Z}_{2^\ell}$-module

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

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

To generate sharings in $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$, we essentially use Protocol 5. However, this protocol does not securely realize $\mathcal{F}_{\mathsf{rand}}$, since in Lemma 3 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 $\llbracket \cdot \rrbracket_{(2\tau)}$, therefore also $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$, does not meet the definition of a secret-sharing scheme in Sect. 2.1). This will only lead to an additive attack in the online phase, which is why we can still use the protocol.

Protocol 7 Secure multiplication up to an additive attack

Inputs: Parties hold correct sharings $\llbracket x \rrbracket_\tau$, $\llbracket y \rrbracket_\tau$.

Preprocessing: The parties execute Protocol 5 for $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$ instead of $\llbracket \cdot \rrbracket_\tau$. They only check correctness for the $\llbracket \cdot \rrbracket_\tau$ part, and not for the $\llbracket \cdot \rrbracket_{(2\tau)}$ part. They obtain a random double sharing $(\llbracket r \rrbracket_\tau, \llbracket r \rrbracket_{(2\tau)})$.

The protocol:

1. The parties locally calculate $\llbracket \delta \rrbracket_{(2\tau)} := \llbracket x \rrbracket_\tau \cdot \llbracket y \rrbracket_\tau - \llbracket r \rrbracket_{(2\tau)}$.
2. Each P_i for $i = 1, \ldots 2\tau{+}1$ sends $u_i := \pi(\lambda_i \delta_i)$ to P_1 (recall $\pi(a_0{+}a_1X{+}\cdots{+}a_{d-1}X^{d-1}) = a_0 \in \mathbb{Z}_{2^\ell}$, 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 $\llbracket x \cdot y \rrbracket_\tau = \llbracket r \rrbracket_\tau + \delta$.

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 $\llbracket x \rrbracket_\tau$, $\llbracket y \rrbracket_\tau$ 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 $\llbracket r \rrbracket_{(\tau,2\tau)}$ is guaranteed to be correct. I.e., the shares are degree τ and 2τ respectively.

Lemma 6. *Protocol 7 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 3. By the proof, we have that $\llbracket r \rrbracket_\tau$ is a correct sharing. The sharing $\llbracket r' \rrbracket_{(2\tau)}$ is not well-defined, because the adversary can change its mind about its shares at any time. However, the adversary always knows the additive error $r' - r$ that it introduces by changing its shares.

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

We distinguish two cases:

- *Case 1: P_1 is not corrupt.* The simulated P_1 receives $\{u_i\}_{i \in \mathcal{C}}$ from $\mathcal{A}$. If it receives $\perp$ for any value u_i, it sends abort to $\mathcal{F}_{\mathsf{mult}}$ and simulates P_1 aborting. Otherwise, it calls $\mathcal{F}_{\mathsf{mult}}$ and receives $\{x_i\}_{i \in \mathcal{C}}, \{y_i\}_{i \in \mathcal{C}}$. For any $i \in \mathcal{C}$, since $\mathcal{S}$ knows x_i, y_i, r'_i, it may calculate $\delta_i = x_i y_i - r'_i$ and thus the value $\pi(\lambda_i \delta_i)$ the adversary is supposed to send if it behaves honestly. The simulator can therefore extract $d = \sum_{i \in \mathcal{C}} u_i - \pi(\lambda_i \delta_i)$. $\mathcal{S}$ does not know the true value of δ, however it may sample $\delta \leftarrow \mathbb{Z}_{2^\ell}$, send it to the corrupted parties, and calculate the corrupted parties' shares as $z_i = r_i + \delta + d$.

  It then simulates the broadcast of δ. If the broadcast aborts, $\mathcal{S}$ simulates the parties aborting and sends abort to $\mathcal{F}_{\mathsf{mult}}$. Otherwise, it sends $d, \{z_i\}_{i \in \mathcal{C}}$ to $\mathcal{F}_{\mathsf{mult}}$, and outputs whatever $\mathcal{A}$ outputs.

  In the ideal execution, $\mathcal{A}$ receives a random δ. It cannot distinguish this from the real value $x \cdot y - r$, since r is uniformly random and by privacy of the secret-sharing scheme it does not have any information on it.
- *Case 2: P_1 is corrupt.* $\mathcal{S}$ samples $\llbracket \delta \rrbracket_{(2\tau)} \leftarrow \llbracket \cdot \rrbracket_{(2\tau)}$. For $i \in \mathcal{H}$ it sends $u_i = \pi(\lambda_i \delta_i)$ to the corrupted P_1. The simulated honest parties receive an identical broadcasted value δ', otherwise the broadcast protocol aborts.

  Since $\mathcal{S}$ knows δ, it can extract $d := \delta' - \delta$, and calculate the corrupted parties' shares as $z_i = r_i + \delta'$. It then sends $d, \{z_i\}_{i \in \mathcal{C}}$ to $\mathcal{F}_{\mathsf{mult}}$, and it outputs whatever $\mathcal{A}$ outputs.

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

In the ideal execution where no abort took place, the actual (non-simulated) parties receive their shares $\{z_i\}_{i \in \mathcal{H}}$ directly from $\mathcal{F}_{\mathsf{mult}}$. The shares are consistent and will reconstruct to the secret $z = x \cdot y + d$. In the ideal execution, the shares are generated by the probabilistic function $\mathsf{share}(z, \{z_i\}_{z \in \mathcal{C}})$, such that the shares are uniformly random subject to the constraints on the shares.[8] In the real execution, the shares also correspond to z. The sharing in the real execution is calculated as $\llbracket r \rrbracket_\tau + \delta$, where $\llbracket r \rrbracket_\tau$ is a uniformly random sharing. Therefore, the outputs are identical in both executions. $\blacksquare$

When evaluating a circuit gate-by-gate using Protocol 7, we consider an optimization in which we do not need to execute the broadcast (which might

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

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 [27]. However, this is not a problem in our case since the degree-2τ sharings have no redundancy in them. As shown in [27], 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.

5.4 Reducing Communication Using Pseudo-Randomness [8,33]

Our protocol as described so far is information-theoretically secure. 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 degree $2t$ 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, so 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 6) for all sharings dealt by P_1 before the verification step in the main protocol.

6 Instantiation for 3 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 Sect. 2.1, and one can efficiently realize the required functionalities described in the same section. Below we discuss some of these properties/functionalities.

6.1 Generating Random Shares

Shares of a random value can be generated non-interactively, as noted in [31,32], 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 straightforward and we omit the details.

6.2 Secure Multiplication up to an Additive Attack

To multiply two secret-shared values, we use the protocol from [4,32], 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 $\sum_{i=1}^{3} \alpha_i = 0$.

The 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 [31]. We note that this can be extended to instantiate $\mathcal{F}_{\mathsf{DotProduct}}$ at the communication cost of one single multiplication, as shown in [14].

6.3 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 1. We remark 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 $\mathcal{F}_{\text{CheckZero}}$ in our compiler is that the parties won't accept a value as 0 when it is not, which is still satisfied by the modified functionality.

Functionality 1 $\mathcal{F}_{\text{RO}}$ – Random Oracle functionality

Setup: Let M be an initially empty map.

The protocol:

- 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.

We have the following proposition.

Proposition 3. *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 parties 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 $v_{i-1} \in \mathbb{Z}_{2^\ell} \backslash \{-(v_i + v_{i+1})\}$ and defines the honest parties' 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. ■

7 Implementation and Evaluation

We implement both protocols in C++ and rely on `uint64_t` and `unsigned int128` types for arithmetic over $\mathbb{Z}_{2^\ell}$, where the former is used when $\ell = 64$ and the latter when $\ell = 128$. 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 $h(X) = X^4 + X + 1$ and denote this by $\mathsf{GR}(2^\ell, d = 4)$. 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 [14]). Concretely, a multiplication in $\mathsf{GR}(2^{64}, 4)$ requires 20 `uint64_t` multiplications and 18 additions, while a multiplication in $\mathbb{Z}_{2^{64}}$ requires 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 over the Galois ring.

Experimental Setup. We run our experiments on AWS `c5.9xlarge` machines, which have 36 virtual cores, 72 GB of memory and a 10 Gbps 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).

7.1 Experiments

For our experiments, we focus on two instantiations.

First we compare our Shamir based instantiation (cf. Sect. 5) against the field protocol of [14]. For this, we use the implementation at [1]. We perform the same benchmarks as reported on in [14]; 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 secret-sharing based instantiation (cf. Sect. 6) against the protocols for computation over rings presented in [21]. 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 [21], we opt instead to use the same experimental setup as theirs in order to obtain a fair comparison. We report here on benchmarks run in a LAN setting. Secondly, we compare our 3-party protocol against the 3-party instantiation from [14]. The 3-party protocol in [14] can be considered state-of-the-art, and thus a comparison against our protocol is in order.

While the protocol of [14] 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 [21] for comparison as their experiments and setup is straightforward to replicate with our protocol, thus allowing us to make a fair comparison. Concurrently with [21], several other proposals for 3 party protocols have been published, such as [13] or [34]. However, no public implementation exists 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 [21]). More precisely, both [13] and [34] evaluate their protocols relative to an implementation of ABY3 [32] that was also implemented by the authors themselves (as no public implementation of ABY3 was available at that time).

While [34] have better amortized communication cost, we estimate that their *concrete* 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 [34] uses the interpolation based check from [8]. For the case of fields, this check was shown in [9] 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 [34] 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.

7.2 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 [14]. However, the overhead is in line 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 motivates a line of research in improving the efficiency of computing over Galois rings, given the relevance of these structures as highlighted in Sect. 1.2. 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 [14].

Interestingly, we see that for a lower number of parties combined with very deep circuits, our protocol performs better in the online phase. E.g., [14] takes 7.3 s, while both of our version is below 4.5 s. This could again be explained by 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 increases, which would be due to the fact that the king needs to send more data during a reconstruction, as well as broadcast being more costly when more parties are involved. We remark that it is possible to distribute the broadcast load of the king among several parties, which may close the gap to some extent.

We see an expected overhead of roughly ×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

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 and b is the time it takes to evaluate the circuit.

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
	[14]	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
	[14]	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
	[14]	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
	[14]	0.38/7.30	0.61/7.32	0.89/8.40	1.05/12.88

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]

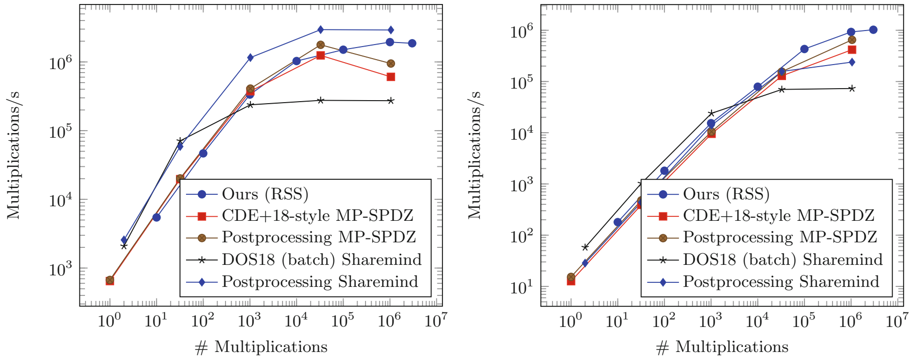
As a more general conclusion, we observe that working over these Galois ring extensions does indeed incur an overhead—even for small extensions such as the one we use.

7.3 Results: Replicated-Based Instantiation

We also compare our replicated secret-sharing based instantiation with the protocols of [21], and present the results in Fig. 1a and Fig. 1b.[10] As we do not have access to the code of all the protocols considered in [21], we run our protocol in the same setup. With the exception of the Sharemind postprocessing protocol, we observe that we outperform all protocols of [21]. 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.

[9] Indeed, while a multiplication in $\mathbb{Z}_{2^{64}}$ is one unsigned 64-bit multiplication, a multiplication on 128-bit types compile to three $\mathbb{Z}_{2^{64}}$ multiplications. That the overhead is less than ×3 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 [21] for giving us the tikzcode of their graph, as well as their raw experimental data which allows us to make a fair comparison in this section.



(a) Throughput in LAN

(b) Throughput in WAN

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

Table 2. LAN times in seconds for circuits with 10^6 multiplications and varying depth with three parties.

	Protocol	20	100	1,000	10,000
RSS	Ours $\ell = 64$	0.23	0.23	0.49	2.36
	Ours $\ell = 128$	0.4	0.41	0.56	2.47
	[14]	0.26	0.33	0.59	2.53

However, when we consider our protocol running in a WAN, we see that we outperform all protocols in [21]. This concurs with the fact that our protocol only needs to send 2 ring elements per multiplication, while the postprocessing protocols of [21] needs to send 3 (Table 2).

We also run our 3-party protocol against the similar field based one from [14]. Given the similarities between the 3-party instantiation in that work and ours, it is not surprising that the two protocols perform very similar. Similar to our Shamir based instantiation, we observe the largest difference (in our favor) with deeper circuits, which we can again attribute to slight differences in the communication layer. On the other hand, the difference is smaller for the more shallow circuits where local computation matters more. For this case, our protocol with $\ell = 64$ is comparable in terms of speed to [14], which uses a 64-bit prime. On the other hand, our protocol with $\ell = 128$ is slightly slower. However, as highlighted in the introduction, the need of an s-gap in field-based protocols to support more complex primitives like secure comparison or truncation implies that comparing a 64-bit prime with $\ell = 64$ is fair.

Acknowledgment. This work has been supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No. 669255 (MPCPRO); and the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC).

References

1. Fast large-scale honest-majority MPC for malicious adversaries (2017). https://github.com/cryptobiu/MPCHonestMajorityNoTriples
2. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Yuan, C.: Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois rings. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019. LNCS, vol. 11891, pp. 471–501. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36030-6_19
3. Araki, T., et al.: Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier, pp. 843–862 (2017)
4. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority, pp. 805–817 (2016)
5. Beerliová-Trubíniová, Z., Hirt, M.: Efficient multi-party computation with dispute control. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 305–328. Springer, Heidelberg (2006). https://doi.org/10.1007/11681878_16
6. Ben-Sasson, E., Fehr, S., Ostrovsky, R.: Near-linear unconditionally-secure multiparty computation with a dishonest minority. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 663–680. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_39
7. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_13
8. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear PCPs. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 67–97. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_3
9. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs, pp. 869–886 (2019)
10. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000)
11. Cascudo, I., Cramer, R., Xing, C., Yuan, C.: Amortized complexity of information-theoretically secure MPC revisited. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 395–426. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_14
12. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 182–199. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15317-4_13
13. Chaudhari, H., Choudhury, A., Patra, A., Suresh, A.: 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, pp. 81–92 (2019)
14. Chida, K., et al.: Fast large-scale honest-majority MPC for malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 34–64. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_2
15. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: SPD$\mathbb{Z}_{2^k}$: efficient MPC mod 2^k for dishonest majority. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 769–798. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_26

16. Dalskov, A.P.K., Escudero, D., Keller, M.: Secure evaluation of quantized neural networks, vol. 2020, no. 4, pp. 355–375 (2020)
17. Damgård, I., Escudero, D., Frederiksen, T.K., Keller, M., Scholl, P., Volgushev, N.: New primitives for actively-secure MPC over rings with applications to private machine learning, pp. 1102–1120 (2019)
18. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_1
19. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74143-5_32
20. Damgård, I., Orlandi, C., Simkin, M.: Yet another compiler for active security or: efficient MPC over arbitrary rings. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 799–829. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_27
21. Eerikson, H., Keller, M., Orlandi, C., Pullonen, P., Puura, J., Simkin, M.: Use your brain! Arithmetic 3PC for any modulus with active security, pp. 5:1–5:24 (2020)
22. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 225–255. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_8
23. Genkin, D., Ishai, Y., Polychroniadou, A.: Efficient multi-party computation: from passive to active security via secure SIMD circuits. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 721–741. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_35
24. Genkin, D., Ishai, Y., Prabhakaran, M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation, pp. 495–504 (2014)
25. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
26. Goldwasser, S., Lindell, Y.: Secure multi-party computation without agreement. J. Cryptol. **18**(3), 247–287 (2005)
27. Goyal, V., Liu, Y., Song, Y.: Communication-efficient unconditional MPC with guaranteed output delivery. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11693, pp. 85–114. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26951-7_4
28. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority MPC. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12171, pp. 618–646. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_22
29. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer, pp. 830–842 (2016)
30. Kikuchi, R., et al.: Field extension in secret-shared form and its applications to efficient secure computation. In: Jang-Jaccard, J., Guo, F. (eds.) ACISP 2019. LNCS, vol. 11547, pp. 343–361. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21548-4_19
31. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority, pp. 259–276 (2017)
32. Mohassel, P., Rindal, P.: ABY^3: a mixed protocol framework for machine learning, pp. 35–52 (2018)

33. Nordholt, P.S., Veeningen, M.: Minimising communication in honest-majority MPC by batchwise multiplication verification. In: Preneel, B., Vercauteren, F. (eds.) ACNS 2018. LNCS, vol. 10892, pp. 321–339. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93387-0_17
34. Patra, A., Suresh, A.: BLAZE: blazing fast privacy-preserving machine learning (2020)