# ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$

Lennart Braun[1], Cyprien Delpech de Saint Guilhem[2], Robin Jadoul[2], Emmanuela Orsini[3], Nigel P. Smart[2,4], and Titouan Tanguy[4]

[1] Department of Computer Science, Aarhus University, Aarhus, Denmark,
[2] COSIC, KU Leuven, Leuven, Belgium,
[3] Department of Computing Sciences, Bocconi University, Milan, Italy,
[4] Zama. Inc, Paris, France.

`braun@cs.au.dk`, `cyprien.delpechdesaintguilhem@kuleuven.be`, `robin.jadoul@esat.kuleuven.be`, `emmanuela.orsini@unibocconi.it`, `nigel.smart@kuleuven.be`, `titouan.tanguy@zama.ai`

**Abstract.** In this work, we extend the MPC-in-the-Head framework, used in recent efficient zero-knowledge protocols, to work over the ring $\mathbb{Z}_{2^k}$, which is the primary operating domain for modern CPUs. The proposed schemes are compatible with any threshold linear secret sharing scheme and draw inspiration from MPC protocols adapted for ring operations. Additionally, we explore various batching methodologies, leveraging Shamir's secret sharing schemes and Galois ring extensions, and show the applicability of our approach in RAM program verification. Finally, we analyse different options for instantiating the resulting ZK scheme over rings and compare their communication costs.

,

# Table of Contents

# 1 Introduction

Zero-knowledge (ZK) proofs [GMR85] are a fundamental tool for numerous privacy-preserving applications. A proof system enables a prover to convince a verifier that a statement is true beyond reasonable doubt. The zero-knowledge property additionally ensures that the only information learnt from the interaction by the verifier (or any other listener) is the veracity of the statement, and nothing else.

A common method of expressing statements for proof systems is circuit satisfiability. In this approach, both the prover and verifier possess a circuit $C$, and the prover aims to demonstrate their knowledge of a witness $w$ which satisfies the condition $C(w) = 0$. Usually, $C$ is a circuit defined over a field, either binary or arithmetic. However, many use cases of ZK proof systems (such as program verification) require the statement to be expressed with arithmetic over a ring, such as $\mathbb{Z}_{2^k}$. In particular, the underlying structure of choice for modern CPUs, 64-bit integers, can be expressed over the ring $\mathbb{Z}_{2^{64}}$. Hence proof systems natively compatible with this ring arithmetic allow to preserve the semantics of a conventional CPU, without the costly need to emulate it with finite field arithmetic instead.

There are few exceptions to this approach and some ZK protocols have been extended to operate over rings. In particular, when considering highly efficient and scalable zero-knowledge protocols, some works [BBMH$^+$21, BBMHS22, LXY23] have extended protocols based on vector oblivious linear evaluation (VOLE) to work over $\mathbb{Z}_{2^k}$. These kinds of proofs are able to handle very large statements, such as proving properties of complex computer programs, but are only designated-verifier, i.e., the verifier needs to keep some state secret from the prover. This means that these proofs cannot be made non-interactive and require both parties to be online at the same time.

Publicly verifiable proofs can be generated in different ways, for example following the MPC-in-the-Head (MPCitH) paradigm introduced by Ishai, Kushilevitz, Ostrovsky and Sahai in [IKOS07]. Despite its simplicity, this technique has proven efficiency and flexibility, and found a variety of different applications. In the context of zero-knowledge, MPCitH leads to very efficient protocols [AHIV17, BN20, GMO16, FR22, FMRV22, KZ22, KKW18] for proving statements that can be expressed with small to medium-size circuits, and it can be used to develop efficient post-quantum digital signature schemes [BDK$^+$21, CDG$^+$17].

*MPC-in-the-Head.* The core idea behind MPCitH is for the prover $\mathcal{P}$ to emulate an MPC protocol for the circuit $C$, amongst $N$ parties, *in their head*, and commit to each of the emulated parties' view. The verifier $\mathcal{V}$ then asks to decommit a small enough subset of these views so as not to break the privacy of the MPC scheme. The soundness of the proof comes from the correctness of the underlying secure MPC protocol and the decommitment of parties' views. In this way, if the prover wants to cheat in the MPC protocol, they need to simulate some parties as acting maliciously, which in turn can be detected if the set of malicious parties overlaps the set of decommited parties. In addition, since the verifier sees fewer views than the privacy threshold of the MPC protocol, the zero-knowledge property holds.

The seminal work of Ishai et al. [IKOS07] describes a generic compiler which makes black-box use of the underlying MPC protocol, but only considers asymptotic complexity; on the other hand, recent concretely efficient protocols [GMO16, FR22, FMRV22, KKW18, AHIV17] provide different concrete instantiations for the MPC protocol used to evaluate the circuit $C$, based both on full-threshold [BN20, KZ22, KKW18, DOT21] and variable $t$-threshold secret-sharing schemes [GMO16, FR22, FMRV22, AHIV17]. In the latter case, the resulting ZK scheme can achieve better soundness and different choices of $t$ result in different proof-size/efficiency/soundness trade-offs.

Another significant difference among these efficient MPCitH based schemes lies in the way the MPC protocol is used, i.e., whether its task consists of *computing* the circuit $C$ or just *verifying* it. In the former approach, taken for example by [BN20, KKW18, IKOS07], the prover locally emulates the MPC protocol by secret-sharing the witness $w$ among the $N$ simulated parties as the input of the MPC evaluation; then it evaluates in MPC the circuit $C$ and sends to the verifier commitments to each parties' input shares, random tapes and received messages (these values constitute a party's *view*) and to all output shares. Then, the verifier randomly chooses $t$ of the views' commitments to be opened, and verifies that the committed messages are all consistent with each other and with the output shares.

In the latter approach, used for example by [AHIV17, BDK$^+$21, DOT21], instead of computing the entire circuit $C$ in MPC, the prover, that knows the witness and all the intermediate values of the circuit evaluation, inputs (or *injects*) all these values (the witness and results of non-linear operations) in a secret-shared form as input of the MPC protocol, whose role at this point is simply checking that these inputs are indeed correct. This approach usually leads to better performance for the prover. The input of this MPC protocol is also called *extended witness*, since the role of the MPC protocol is not only that of verifying that $w$ is a valid witness, i.e., that $C(w) = 0$, but also that the non-linear operations in $C$ have been honestly computed.

### 1.1 Our Contribution

This work describes how to adapt some efficient MPCitH protocols, like [BN20, DOT21, FR22], to work over a ring of the form $\mathbb{Z}_{2^k}$. As said before, compared to VOLE-based schemes, MPCitH proofs have the advantage to be public coin, which enables public verifiability and the ability to obtain non-interactive proofs via the Fiat–Shamir transformation [FS87].[5] We summarize our contributions as follows.

*MPCitH over* $\mathbb{Z}_{2^k}$. Our approach considers MPCitH schemes such as Limbo [DOT21] and [FR22] where the MPC protocol is used to *verify* the correctness of the committed extended inputs. This type of protocols can be well suited to particular use cases, such as verifying computations or proving the correct execution of RAM programs (where an extension of existing protocols to work over $\mathbb{Z}_{2^k}$ can be practically relevant).

In recent years, MPC protocols have also been extended to work over rings; see for example [CDE$^+$18, EXY22] for the case of dishonest majority (i.e. $t \geq N/2$), and [ACD$^+$19, JSL22] for the case of honest majority (i.e. $t < N/2$). In the case of honest majority protocols, the natural secret-sharing scheme to instantiate a threshold MPC protocol, Shamir's secret sharing [Sha79], requires the underlying algebraic structure to be suitably large. In the case of MPC over finite fields one simply extends the base field so that it contains $N + 1$ elements (where $N$ is the number of parties). In the case of rings it requires a large enough Galois ring extension, so that the largest *exceptional sequence*[6] in the extension ring contains $N + 1$ elements. This was originally introduced in the context of secret sharing by Fehr [Feh98].
A similar approach is also needed in our protocols, where we replace the full-threshold additive sharing scheme used in Limbo with a $t$-threshold secret sharing scheme to achieve better soundness. We show different options to instantiate our MPC verification procedures, and analyse their respective communication costs. While the $t$-threshold approach generally comes with a larger proof size than the additive sharing, it trades this for higher efficiency for the verifier, who now only needs to verify that $t$ parties behaved honestly rather than $N - 1$.

[5] Many VOLE proofs can be split into an interactive, witness-independent preprocessing phase and a public-coin online phase, of which the latter can be made non-interactive. Note that this still requires the designated verifier to keep secret state.

[6] Informally, an exceptional sequence of elements in a ring $R$ is such that their pairwise difference is invertible. (See Section 2.2.)

Finally, we recall that KKW [KKW18] already works over any rings. This scheme is known for its efficiency when dealing with small to medium-sized circuits, however, as mentioned earlier, it requires an MPC evaluation of the entire circuit $C$, which may not be the most suitable approach for applications like program verification.

*Packing techniques.* In Section 6, we describe a methodology for *packing* within our MPCitH proofs, that is, proving multiple statements for the same circuit in parallel, in a single proof. It consists of two orthogonal approaches that could potentially be combined to achieve better packing rates. We take advantage of Shamir's threshold secret sharing scheme by embedding multiple secrets in the roots of the sharing polynomial, and we also make use of the additional coefficients provided by Galois ring extensions by placing multiple secrets within a single ring element.

Performing batch proofs in this way additionally alleviates the extra communication cost for a threshold scheme, since the extra space that was introduced to have a large enough exceptional set becomes completely utilised. In combination with the increased verifier efficiency and the better soundness guarantees, this makes the threshold setting preferable to the additive setting for batch proofs.

*RAM applications.* In Section 7, we adapt the compilation procedure of [DOTV22] to the ring structure. The techniques used there allow to *compile* a list of read and write array accesses to a *standard* arithmetic circuit for proof systems in order to enable program verification. This compilation naturally fits the MPCitH framework extended to the ring $\mathbb{Z}_{2^k}$ that we describe in this paper. This approach removes the need of any bit-decomposition operation; this is different from other recent works [GHAH$^+$23] that use MPCitH schemes based on the KKW protocol [KKW18] for program verification and ring switching techniques based on edaBits [EGK$^+$20].

In our work, to verify the correctness of the memory operations, the initial array is extended to a *checking circuit* $C_{\mathsf{check}}$ over $\mathbb{Z}_{2^k}$—with standard linear and multiplication gates and calls to a random oracle—that verifies the consistency of a list of access tuples which contains both the initial array and the accesses performed, encoded as a set of tuples. Given this list, $C_{\mathsf{check}}$ produces new multiplication triples that need to be verified via a checking procedure over rings. To perform these consistency checks, [DOTV22] describes three subcircuits $\mathsf{EqCheck}$, $\mathsf{BdCheck}$ and $\mathsf{PermCheck}$ to verify respectively equality, upper and lower bounds and permutation of a list of values in zero-knowledge.

While our compilation follows the blueprint of [DOTV22], the main difference is that, to suit the ring structure, we require a large enough exceptional sequence and the removal of the $\mathsf{EqCheck}$ sub-circuit that crucially relies on every element having an inverse. Our resulting construction inherits all the properties of the scheme described in [DOTV22], leading to a public-coin constant-overhead ZK proof system for computations over $\mathbb{Z}_{2^k}$ in the RAM model.

## 2 Preliminaries

This section establishes notation and recalls standard results.

### 2.1 Notation

We denote by $\lambda$ the computational security parameter and by $\sigma$ the statistical security parameter. For a set $S$, we let $a \leftarrow S$ denote the uniform sampling $a$ from $S$. If $D$ is a probability distribution over $S$, we let $a \leftarrow D$ denote sampling $a$ from $S$ according to $D$. For a probabilistic algorithm $A$, we let $a \leftarrow A$ denote the probabilistic assigning to $a$ of the output of algorithm $A$, with the distribution being determined by the random coins of $A$. We let $[n] \subset \mathbb{N}$ denote the set $\{1, \ldots, n\}$. We use $\mathbf{x}$ for vectors of elements, and $\mathbf{x} \circ \mathbf{y}$ for element-wise products.

*Zero-knowledge proofs.* We use standard definitions of zero-knowledge proofs; we construct our protocols to allow proving arbitrary NP language-membership statements. Let $L$ be in NP and $\mathcal{R}(x, w)$ be a corresponding NP relation with statement $x$ and witness $w$. That is, the statement $x$ is a member of $L$ if and only if a witness $w$ exists such that $(x, w) \in \mathcal{R}$. We can then consider an arithmetic circuit $C$ (with addition and multiplication gates) that decides (or rather confirms) membership of $L$ when given such a witness. Concretely, the circuit satisfies $C(x, w) = 0$ if and only if $(x, w) \in \mathcal{R}$. The focus of this work are zero-knowledge proofs of knowledge for relations where $C$ is an arithmetic circuit over the ring $\mathbb{Z}_{2^k}$.

### 2.2 Rings

While the circuits we use in our proof systems are defined over the ring $\mathbb{Z}_{2^k}$, we need to work over larger rings to enable threshold secret sharing and to achieve low soundness errors. In this work we consider two ways to obtain such larger rings as described below.

**2-adic extensions.** Instead of using $\mathbb{Z}_{2^k}$, we increase the modulus and work over $\mathbb{Z}_{2^{k+s}}$, where $s$ depends on the security parameter. This methodology of extending the ring 2-adically in order to check various relations was first introduced in the SPD$\mathbb{Z}_{2^k}$ protocol [CDE+18]. While this is a well-studied technique in the MPC literature, there are some limitations inherent to our application to MPCitH. Many soundness checks that use such an extension only guarantee consistency for the $k$ lower bits; this may therefore require iterating such extensions to $\mathbb{Z}_{2^{k+n\cdot s}}$. Moreover, since $\mathbb{Z}_{2^k}$ is not a subring of $\mathbb{Z}_{2^{k+s}}$, we cannot easily lift $\mathbb{Z}_{2^k}$ elements to $\mathbb{Z}_{2^{k+s}}$ if we also wish to retain some auxiliary algebraic relationship between the lifted values. The converse direction—truncating elements of $\mathbb{Z}_{2^{k+s}}$ to $\mathbb{Z}_{2^k}$—is a well-defined ring homomorphism.

**Galois extensions.** We extend the base ring $\mathbb{Z}_{2^k}$ by forming the Galois ring $GR(2^k, d) = \mathbb{Z}_{2^k}[X]/(p(X))$, the ring of polynomials with $\mathbb{Z}_{2^k}$ coefficients reduced modulo an irreducible polynomial $p(X)$ of degree $d$. One advantage of this technique is that reduction modulo 2 results in the field $\mathbb{F}_{2^d}$, i.e., we have $GR(2^k, d)/(2) \simeq \mathbb{F}_{2^d}$. Also, while taking a degree-$d$ extension increases the size of elements by a multiplicative factor $d$, it can be used for several different checks—unlike the 2-adic extensions. Moreover, a $\mathbb{Z}_{2^k}$ element can be easily "lifted" into a $GR(2^k, d)$ element by using zero for the coefficients of non-constant terms. This lift often retains algebraic relationships between the lifted elements.

Note that both techniques can also be combined to obtain rings of the form $GR(2^{k+s}, d)$.

**Definition 2.1 ((Maximal) Exceptional Sequence).** *Let $GR(2^k, d)$ be a degree-$d$ Galois extension of $\mathbb{Z}_{2^k}$. A set $\{\alpha_1, \ldots, \alpha_n\}$ is an* exceptional sequence (of length $n$) *in $GR(2^k, d)$ if for all $i \neq j \in [n]$ we have $\alpha_i - \alpha_j \in GR(2^k, d)^*$.*

*An exceptional sequence of length $n$ is* maximal *if there does not exist an exceptional sequence of length $n' > n$.*

In $GR(2^k, d)$, there exists a maximal exceptional sequence of length $2^d$, see [ACD+19, Prop. 2]. We use $\mathsf{Ex}(R)$ to denote a maximal exceptional sequence of a Galois ring $R$ and assume that we can efficiently sample uniformly random elements from it. For $\mathsf{Ex}(R)$ we can take the $2^d$ polynomials with $\{0, 1\}$ coefficients as an exceptional sequence.

To perform soundness checks in our proof systems, we will often reduce these to equality checks between two polynomials. While the Schwartz–Zippel Lemma is frequently used for this purpose when the polynomials are defined over finite fields, we require a generalised variant that is adapted to our ring-based setting.

**Lemma 2.1 (Generalized Schwartz–Zippel Lemma [CCKP19]).** *Let $R$ be a ring, and $f\colon R^n \to R$ an $n$-variate non-zero polynomial of total degree (the sum of degrees of each variable) $D$ over $R$. Let $A \subseteq R$ be a finite exceptional sequence with $|A| \geq D$. Then,* $\Pr_{\mathbf{x} \in_R A^n}[f(\mathbf{x}) = 0] \leq \frac{D}{|A|}$.

For soundness checks over 2-adic extensions, we also introduce the following lemma to bound the soundness error over $\mathbb{Z}_{2^k}$ when performing computations over $\mathbb{Z}_{2^{k+s}}$.

**Lemma 2.2 (2-adic Random Linear Combinations).** *Let $\delta_1, \ldots, \delta_n$ be elements of $GR(2^{k+s}, d)$, such that at least one $\delta_i \not\equiv 0 \pmod{2^k}$. Also let $\alpha_1 = 1$ and $\alpha_2, \ldots, \alpha_n \leftarrow GR(2^{s+1}, d)$ be chosen uniformly at random. Then we have the probability bound* $\Pr\left[\sum \alpha_i \cdot \delta_i \equiv 0 \pmod{2^{k+s}}\right] \leq 2^{-(s+1)\cdot d}$.

*Proof.* Let $\delta_j$ (for $j \neq 1$)[7] be a value that is nonzero modulo $2^k$ and $w < k$ be the maximal integer such that $2^w \mid \delta_j$. Then $\sum \alpha_i \cdot \delta_i \equiv 0 \pmod{2^{k+s}}$ only when

$$\alpha_j \equiv \frac{-\sum_{i \neq j} \alpha_i \cdot \delta_i}{2^w} \cdot \left(\frac{\delta_j}{2^w}\right)^{-1} \pmod{2^{k+s-w}},$$

where the inverse used is guaranteed to exist due to the maximality of $w$. Since $\alpha_j$ is uniformly random from $GR(2^{s+1}, d)$ and $k + s - w \geq s + 1$, our claim holds. □

### 2.3 Secret-Sharing Schemes over Rings

We consider additive ($A$) as well as threshold ($T$) secret sharing schemes over our commutative finite rings $R$, e.g. $R = GR(2^k, d)$, which we denote as $[\![\cdot]\!]^A$ and $[\![\cdot]\!]^T$ respectively. Our protocols work with any *linear* secret sharing scheme. Only the overall soundness and the communication cost depend on the instantiation. Hence, we will often drop the $A$ or $T$ from the notation and just write $[\![\cdot]\!]$. Both schemes allow the parties to compute linear functions on shared values such as $[\![\gamma]\!] = a \cdot [\![\alpha]\!] + b \cdot [\![\beta]\!] + c$ by performing only local computations on their individual shares.

**Additive Secret-Sharing.** An additive $(N-1)$-out-of-$N$ secret sharing over $R$ is straightforward. To share a value $v \in R$, first sample values $v_1, \ldots, v_N \leftarrow R$ and then set $\Delta_v = v - \sum_{i \in [N]} v_i$. The share of party $P_i$ is then defined as $[\![v]\!]^A_i := (v_i; \Delta_v)$. We denote this procedure as $[\![v]\!]^A \leftarrow \mathsf{Share}^A(v)$. Reconstruction is performed by computing $v = \Delta_v + \sum_{i \in [N]} v_i$, which we denote as $v \leftarrow \mathsf{Rec}^A([\![v]\!]^A)$.

**Threshold Secret-Sharing.** The well-known threshold secret sharing scheme due to Shamir [Sha79] relies on polynomial interpolation which usually requires a field structure. We follow the work of Abspoel et al. [ACD⁺19], who have shown how to use Galois rings to realize Shamir-style threshold secret sharing over rings in the context of MPC.

Let $\alpha_0, \ldots, \alpha_N$ be an exceptional sequence of length $N+1$ within $GR(2^k, d)$. To share a value $v \in \mathbb{Z}_{2^k}$ among parties $P_1, \ldots, P_N$ with threshold $t$, first sample a random degree-$t$ polynomial $f$ from $GR(2^k, d)[X]^{\leq t}$ with the condition that $f(\alpha_0) = v$. To then create shares, give each party $P_i$, for $i \in [N]$, the value $[\![v]\!]^T_i := y_i := f(\alpha_i)$. We denote such a sharing with $[\![v]\!]^T \leftarrow \mathsf{Share}^T(v)$.

To reconstruct a value $v$, we use Lagrange interpolation using any index set $S \subseteq [1, N]$ of at least $t + 1$ shares:

$$f(X) = \sum_{i \in S} y_i \cdot \prod_{j \in S \setminus \{i\}} \frac{X - \alpha_j}{\alpha_i - \alpha_j}$$

[7] if only $\delta_1 \not\equiv 0$, the equality holds with probability 0.

This interpolation over $GR(2^k, d)$ is well-defined since, by definition of an exceptional sequence, all differences $\alpha_i - \alpha_j$ are invertible. Let the reconstruction procedure be denoted by $v \leftarrow \mathsf{Rec}^T(\{[\![v]\!]_i^T\}_{i\in S})$.

Note that, in general, one needs to check whether a shared value lies in the base ring $\mathbb{Z}_{2^k}$ or (strictly) in the ring extension $GR(2^k, d) \setminus \mathbb{Z}_{2^k}$. To deal with this, we describe a checking procedure $\Pi_{\mathsf{Ring\text{-}Check}}$, which ensures a set of shares corresponds to values in $\mathbb{Z}_{2^k}$ without violating $t$-privacy, in Section 4. This procedure can then be applied to the input shares. In our protocols, no other wires or shares, such as the rest of the extended witness, need be validated in this way, as either these shares are obtained through linear operations that preserve this property, or the property is guaranteed by the correctness of our subprotocol to check multiplications.

### 2.4 MPC-in-the-Head via Linear Secret Sharing

This section presents a general framework for MPCitH protocols based on threshold linear secret sharing schemes, built on the framework of Feneuil et al. [FR22] that provides a generic transformation for MPC protocols based on threshold linear secret sharing. We first describe a generic MPC protocol for circuit verification, then show how it can be used to obtain a ZK proof system, and finally analyse the resulting soundness.

**MPC Protocol for MPCitH.** The MPC protocol presented in Figure 1 is generic for threshold LSSS over $\mathbb{Z}_{2^k}$, in the sense that it can be instantiated with any *multiplication checking protocol* and any suitable LSSS. It involves an *input party* who distributes secret shared values to the computing parties. Looking ahead, we refer to the totality of these input values as the *extended witness* of the resulting proof system. In addition, computing parties have access to two oracles: a *hint oracle* $\mathcal{O}_H$ which provides the parties with a sharing of an arbitrary secret value from the input party; and a *randomness oracle* $\mathcal{O}_R$ which outputs random public values.

These oracles are mainly used in the following subprotocols whose goal is to verify some properties on shares of (extended) witness values:

$\Pi_{\mathsf{Zero\text{-}Check}}$ takes as input a value $[\![v]\!]$ (resp. a vector of values $[\![\mathbf{v}]\!]$) and returns $\top$ when $v = 0$ (resp. every entry of $\mathbf{v}$ is zero), or $\perp$ otherwise. This can be achieved similarly to share reconstruction, with the difference that the opened value is not sent.

$\Pi_{\mathsf{Mult\text{-}Check}}$ takes a triple $([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ and returns $\top$ if and only if $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$. In some cases, this equality can be checked over a different ring than that in which the input values are shared. We provide three instantiations of $\Pi_{\mathsf{Mult\text{-}Check}}$ in Section 3, and these form the main contribution of this paper.

$\Pi_{\mathsf{Ring\text{-}Check}}$ takes as input a vector of values $[\![\mathbf{v}]\!]$, shared over a 2-adic extension $GR(2^{k+s_{\mathsf{rc}}}, d_0)$ and outputs $\top$ if and only if the truncation of $\mathbf{v}$ to $GR(2^k, d_0)$ lies in the subring $\mathbb{Z}_{2^k}$. It also truncates the elements of $\mathbf{v}$ to the ring $GR(2^{k+s}, d_0)$. (See Section 4.)

We write $\Pi^{\tau}_{\mathsf{Mult\text{-}Check}}$ to denote the parallel repetition of $\tau$ instances. By *verifying* a property through one of these subprotocols, we mean that the subprotocol is run, and $\mathsf{Reject}$ is returned by the MPC protocol when the output differs from $\top$. Reconstructing a shared value is performed by each party $P_j$ first broadcasting its share $[\![v]\!]_j$ and then running $v \leftarrow \mathsf{Rec}([\![v]\!])$ In the threshold setting, only $t + 1$ shares are required since the other shares are determined by these.

In essence, this protocol does not compute the circuit $C$, but only checks that the values given by the input party are consistent with an honest evaluation of $C$. To do so, the computation parties parse $C$ in topological order but only (locally) compute the linear gates, whereas output of non-linear gates and $\mathsf{Rec}$ are provided as input and hence need to be checked. This is necessary because the input party is not trusted to provide the correct values. The output of the protocol

**Generic MPC Protocol $\Pi_C$ for Circuit Verification**

**Parameters:** A circuit $C$ over $\mathbb{Z}_{2^k}$ consisting of linear and multiplication gates with $\#\mathsf{inputs}$ inputs and $m$ multiplications $\mathsf{Mul}$; a LSSS sharing scheme $[\![.]\!]$ defined over $GR(2^{k+s}, d_0)$ for parameters $s$ and $d_0$. The inputs $w_i$ are defined over $GR(2^{k+s_{\mathsf{rc}}}, d_0)$, for parameter $s_{\mathsf{rc}} \geq s$ which matches the parameter for $\Pi_{\mathsf{Ring\text{-}Check}}$.

**Inputs:** The input party calls $\mathsf{Share}$ on its input $w_i$, $i \in [\#\mathsf{inputs}]$ and $w_\gamma$ for each gate $(\alpha, \beta, \gamma, \mathsf{Mul})_i$ for $i \in [m]$, and send $[\![w_*]\!]_j$ to the computing party $P_j$.

**Protocol:** Each $P_j$ initializes an empty checklist $\mathcal{M}$

1. Verify the inputs are in $\mathbb{Z}_{2^k}$: $\Pi_{\mathsf{Ring\text{-}Check}}(w_1, \ldots, w_{\#\mathsf{inputs}})$
2. For each gate $(\alpha, \beta, \gamma, T) \in C$, in topological order:
   (a) Case $T = \mathsf{Lin}$: $[\![v_\gamma]\!] := a \cdot [\![v_\alpha]\!] + b \cdot [\![v_\beta]\!] + c$ done locally by each party.
   (b) Case $T = \mathsf{Mul}$:
      – Party $P_j$ retrieves $[\![w_\gamma]\!]_j$ received from the input party and sets $[\![v_\gamma]\!]_j = [\![w_\gamma]\!]_j$.
      – Each $P_j$ adds a tuple to (their share of) the multiplication checklist $\mathcal{M}_j \leftarrow \mathcal{M}_j \cup \{([\![v_\alpha]\!]_j, [\![v_\beta]\!]_j, [\![v_\gamma]\!]_j)\}$
3. Verify circuit output: $\Pi_{\mathsf{Zero\text{-}Check}}([\![v_o]\!])$.
4. Verify multiplications: parties parse $\mathcal{M}$ column-wise as $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ and run $\Pi^{\tau_{\mathsf{in}}}_{\mathsf{Mult\text{-}Check}}([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$.

**Fig. 1.** Generic MPC protocol for circuit verification

is either $\mathsf{Accept}$ or $\mathsf{Reject}$. To decrease the false-positive rate of the multiplication checking procedure, the parties execute it $\tau_{\mathsf{in}}$ times in parallel.

**From MPC to ZK.** The compilation technique of Ishai et al. [IKOS07], applied to this MPC protocol, provides our interactive zero-knowledge scheme between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

The prover executes, in their head, the MPC protocol $\Pi_C(x, w)$ between $N$ parties using an LSSS with $t$-privacy. To do so, $\mathcal{P}$ first evaluates $C(x, w)$ in the clear, and secret shares $w$ as well as the intermediate values required for a local computation of $C$. After recording these $N$ input views, it plays the role of the input party and distributes these shares to virtual computing parties. These parties execute $\Pi_C(x, w)$ and its checking sub-protocols. When the protocol queries $\mathcal{O}_H$, the requested shared values are provided by $\mathcal{P}$ to the virtual parties and recorded in the input views. Queries to $\mathcal{O}_R$ are replaced by an interaction with the verifier, where first $\mathcal{P}$ commits to the input views so far, and then $\mathcal{V}$ responds with a random value.

In the final interaction, after $\Pi_C$ terminates, $\mathcal{V}$ asks to open $t$ of the $N$ views, which it checks for consistency. If the consistency check succeeds, and the output of $\Pi_C(x, w)$ is $\mathsf{Accept}$, then $\mathcal{V}$ also outputs $\mathsf{Accept}$.

**ZK Protocol Soundness.** The MPC protocol may output $\mathsf{Accept}$ for an invalid witness with some bounded false-positive rate $p$, i.e., the probability that $\Pi_C(x, w)$ outputs $\mathsf{Accept}$ when in fact $C(x, w) \neq 0$. When $p$ is not sufficiently small, we increase the detection probability by performing $\tau_{\mathsf{in}}$ parallel *inner repetitions* of the circuit check *inside* the MPC protocol. This leads to an overal false-positive rate of $\mathsf{err}_{\mathsf{MPC}} = p^{\tau_{\mathsf{in}}}$.

The framework of Feneuil et al. [FR22] provides a generic transformation for any such MPC protocol with $N$ parties and tolerating up to $t$ corruptions into an MPCitH proof, with a soundness error of

$$\mathsf{err}_{\mathsf{ZK}} = \frac{1}{\binom{N}{t}} + \mathsf{err}_{\mathsf{MPC}} \cdot \frac{t \cdot (N-t)}{t+1}. \tag{1}$$

For an additive full-threshold secret sharing scheme ($t = N - 1$), this becomes

$$\mathsf{err}_{\mathsf{ZK}} = \frac{1}{N} + \mathsf{err}_{\mathsf{MPC}} \cdot \left(1 - \frac{1}{N}\right).$$

By setting $N$ and $t$, we obtain a certain $\mathsf{err}_{\mathsf{ZK}}$ for the soundness error of a *single execution* of the protocol. Since this may be too high for a given security setting, we can repeat the transformed protocol $\tau_{\mathsf{out}}$ times (*outer repetitions*) to obtained any desired soundness error, $\mathsf{err}_{\mathsf{ZK}}^{\tau_{\mathsf{out}}}$.

We denote the overall proof size by $\mathsf{size}_{\mathsf{Proof}}$, which one can think of as the communication cost in bits, required to commit to the parties' views and open $t$ of them in $\tau_{\mathsf{out}}$ repetitions.

## 3 Checking Multiplications over Rings

We now describe three instantiations for $\Pi_{\mathsf{Mult\text{-}Check}}$. The three protocols have appeared previously in the context of MPCitH over fields, but their extension to MPC over rings is mostly new, although a protocol similar to our sacrificing check can be found in [BBMH+21] for VOLE-based zero-knowledge proofs over $\mathbb{Z}_{2^k}$.

We analyse their soundness in the ring-based setting, and compare their performance. For each of the checking procedures, we analyse the false-positive rate $\mathsf{err}_{\mathsf{MPC}}$ of the resulting MPC protocol. It then suffices to use the generic transformation of Feneuil and Rivain [FR22] to compile our MPC protocol into an MPCitH proof system with soundness error as in eq. (1).

Our three different checking procedures are: 1) A simple sacrifice-based check, $\Pi_{\mathsf{Sac\text{-}Check}}$ (described in Section 3.1), 2) an inner product multiplication check, $\Pi_{\mathsf{IP\text{-}Check}}$ (in Section 3.2), and 3) a compressed multiplication check, $\Pi_{\mathsf{Compress}}$ (in Section 3.3). For the first two of these, one can improve the soundness by utilizing either 2-adic or Galois extensions. The third, compressed multiplication check, is adapted from the methodology in [BBC+19, DOT21], and requires a Galois ring extension.

Looking ahead, in the next section we also present a fourth procedure which checks that a set of shares (typically the input to the circuit) all correspond to values in $\mathbb{Z}_{2^k}$ (as in line 1 of Figure 1). This procedure takes its inputs as shares in $GR(2^{k+s_{\mathsf{rc}}}, d_0)$, has a soundness error of $\mathsf{err}_{\mathsf{Ring\text{-}Check}}$. When the chosen multiplication checking procedure would have sufficient soundness with smaller $s < s_{\mathsf{rc}}$, it is possible to locally truncate the input shares correspondingly before performing the procedure.

The false-positive rate of the MPC protocol becomes $\mathsf{err}_{\mathsf{MPC}} := \mathsf{err}_{\mathsf{Check}}^{\tau_{\mathsf{in}}} + \mathsf{err}_{\mathsf{Ring\text{-}Check}}$ where $\mathsf{err}_{\mathsf{Check}}$ denotes the false-positive rate of a single execution of the checking procedure. In Section 5, we investigate the differences in communication cost for our different multiplication checks and sharing scheme choices.

### 3.1 Sacrifice Based Check

Our first multiplication checking procedure is a sacrificing based check. This is based on the checking protocol of Baum and Nof [BN20], combined with an optimization of Kales and Zaverucha [KZ22, Sec. 2.5, Optimization 3], transferred to the ring setting. The algorithm is presented in Figure 2.

As inputs, it receives the vectors $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ of multiplication input and output values, secret-shared over the "computation ring" $GR(2^{k+s}, d_0)$. In case of $d_1 > 1$, it first lifts these vectors to the "checking ring" $GR(2^{k+s}, d_0 \cdot d_1)$. Then, the hint oracle $\mathcal{O}_H$ distributes to the parties secret shares of $[\![\mathbf{a}]\!]$ and $[\![\mathbf{c}]\!]$, correlated in such a way that $\mathbf{a} \circ \mathbf{y} = \mathbf{c}$. After receiving a random coefficient $\varepsilon$ from the randomness oracle $\mathcal{O}_R$, the parties "sacrifice" the vector $[\![\mathbf{a}]\!]$ by using it to mask the randomized vector $\varepsilon \cdot [\![\mathbf{x}]\!]$ and reconstruct the masked value as $\boldsymbol{\alpha}$. Finally, the protocol checks whether both $\mathbf{z}$ and $\mathbf{c}$ were computed correctly by $\mathcal{O}_H$ by checking that the sacrificing equation $\varepsilon \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] - \boldsymbol{\alpha} \circ [\![\mathbf{y}]\!]$ is equal to 0. The argument is that if either $\mathbf{z}$ or $\mathbf{c}$ is incorrect, then the probability that the equality holds, taken over the choice of $\varepsilon \in GR(2^{1+s}, d_0 \cdot d_1)$, is very small.

**$\Pi_{\text{Sac-Check}}$: Sacrificing Check**

**Parameters:** Additional Galois extension size $d_1$.

**Inputs:** $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ shared over $GR(2^{k+s}, d_0)$.

**Protocol:**

1. Lift $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ to $GR(2^{k+s}, d_0 \cdot d_1)$.
2. $(\llbracket \mathbf{a} \rrbracket, \llbracket \mathbf{c} \rrbracket) \leftarrow \mathcal{O}_H$ uniformly random with $\mathbf{a} \circ \mathbf{y} = \mathbf{c}$ over $GR(2^{k+s}, d_0 \cdot d_1)$
3. $\varepsilon \leftarrow \mathcal{O}_R$ such that $\varepsilon \in GR(2^{1+s}, d_0 \cdot d_1)$
4. $\boldsymbol{\alpha} \leftarrow \mathsf{Rec}(\varepsilon \cdot \llbracket \mathbf{x} \rrbracket - \llbracket \mathbf{a} \rrbracket)$
5. Output $\Pi_{\text{Zero-Check}}(\varepsilon \cdot \llbracket \mathbf{z} \rrbracket - \llbracket \mathbf{c} \rrbracket - \boldsymbol{\alpha} \circ \llbracket \mathbf{y} \rrbracket)$

**Fig. 2.** The sacrificing check over rings.

We first take a brief look at the correctness of the protocol. If the input is valid, then the protocol always outputs $\mathsf{Accept}$, since

$$\begin{aligned}\varepsilon \cdot \mathbf{z} - \mathbf{c} - \boldsymbol{\alpha} \circ \mathbf{y} &= \varepsilon \cdot \mathbf{x} \circ \mathbf{y} - \mathbf{a} \circ \mathbf{y} - (\varepsilon \cdot \mathbf{x} - \mathbf{a}) \circ \mathbf{y} \\ &= \varepsilon \cdot \mathbf{x} \circ \mathbf{y} - \mathbf{a} \circ \mathbf{y} - \varepsilon \cdot \mathbf{x} \circ \mathbf{y} + \mathbf{a} \circ \mathbf{y} = 0.\end{aligned}$$

The zero-knowledge property remains preserved by virtue of $\boldsymbol{\alpha}$ being uniformly random as a result of the mask $\mathbf{a}$ being uniformly random.

Soundness follows from the following theorem.

**Theorem 3.1 (Soundness of $\Pi_{\text{Sac-Check}}$).** *For invalid input, i.e., $\exists i \in [m]$ . $x_i \cdot y_i \neq z_i$, the check passes with probability at most* $\mathsf{err}_{\text{Sac-Check}} := 2^{-(s+1)\cdot d_0 \cdot d_1}$.

*Proof.* Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ and $\mathbf{a} \circ \mathbf{y} = \mathbf{c} + \boldsymbol{\delta}_c$. The protocol outputs $\mathsf{Accept}$ if and only if for all $i \in [m]$, we have

$$\begin{aligned}0 &= \varepsilon \cdot z_i - c_i - \alpha_i \cdot y_i \\ &= \varepsilon \cdot (x_i \cdot y_i + \delta_{z,i}) - (a_i \cdot y_i + \delta_{c,i}) - (\varepsilon \cdot x_i - a_i) \cdot y_i \\ &= \varepsilon \cdot x_i \cdot y_i + \varepsilon \cdot \delta_{z,i} - a_i \cdot y_i - \delta_{c,i} - \varepsilon \cdot x_i \cdot y_i + a_i \cdot y_i \\ &= \varepsilon \cdot \delta_{z,i} - \delta_{c,i}.\end{aligned}$$

Recall that $\varepsilon \in_R GR(2^{s+1}, d_0 \cdot d_1)$, $\delta_{z,j} \in GR(2^{k+s}, d_0)$, and $\delta_{c,j} \in GR(2^{k+s}, d_0 \cdot d_1)$. Assume that $\delta_{z,j} \neq 0 \pmod{2^k}$ for some $j \in [m]$. By Lemma 2.2, we can bound the probability that a malicious prover chooses $\delta_{z,j}, \delta_{c,j}$ such that $0 = \varepsilon \cdot \delta_{z,j} + \delta_{c,j}$ holds over $GR(2^{k+s}, d_0 \cdot d_1)$. $\square$

### 3.2 Inner Product Multiplication Check

Our second checking procedure, which is based on inner product checks, is described as a precursor to the Limbo protocol [DOT21], together with optimizations from Kales and Zaverucha [KZ22], adapted to the ring setting. We present the algorithm in Figure 3.

This second checking procedure $\Pi_{\text{IP-Check}}$ works very similarly to the sacrificing check $\Pi_{\text{Sac-Check}}$ of Figure 2, the main difference is that the hint oracle $\mathcal{O}_H$ produces a single correlated inner product tuple $((\mathbf{a}, c)$ such that $\langle \mathbf{a}, \mathbf{y} \rangle = c)$ rather than $m$ correlated multiplication tuples $((\mathbf{a}, \mathbf{c})$ such that $\mathbf{a} \circ \mathbf{y} = \mathbf{c})$. This change then requires the random oracle $\mathcal{O}_R$ to produce $m$ random values (contained in the vector $\boldsymbol{\eta}$), instead of a single one, and it also changes the checking equation so that it checks a single equality, rather than $m$. This time, the security rationale is that if either $\mathbf{z}$ or $c$ is incorrect, then the single checking equation will not equal 0 except with small probability (over the choice of $\boldsymbol{\eta}$). The rationale for the zero-knowledge property is again due to the random mask $\llbracket \mathbf{a} \rrbracket$.

**$\Pi_{\mathsf{IP\text{-}Check}}$: Inner Product Check**

**Parameters:** Additional Galois extension size $d_1$.

**Inputs:** $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ shared over $GR(2^{k+s}, d_0)$.

**Protocol:**

1. Lift $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ to $GR(2^{k+s}, d_0 \cdot d_1)$.
2. $([\![\mathbf{a}]\!], [\![c]\!]) \leftarrow \mathcal{O}_H$ uniformly random with $\langle \mathbf{a}, \mathbf{y} \rangle = c$ over $GR(2^{k+s}, d_0 \cdot d_1)$.
3. $\boldsymbol{\eta} \leftarrow \mathcal{O}_R$ such that $\boldsymbol{\eta} \in GR(2^{1+s}, d_0 \cdot d_1)^m$.
4. $\boldsymbol{\alpha} \leftarrow \mathsf{Rec}(\boldsymbol{\eta} \circ [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!])$
5. Output $\Pi_{\mathsf{Zero\text{-}Check}}(\langle \boldsymbol{\eta}, [\![\mathbf{z}]\!] \rangle - [\![c]\!] - \langle \boldsymbol{\alpha}, [\![\mathbf{y}]\!] \rangle)$

**Fig. 3.** The inner product check over rings.

Here as well, the protocol is correct, since if the input is valid, then the protocol always outputs $\mathsf{Accept}$ as

$$\begin{aligned}\langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\alpha}, \mathbf{y} \rangle &= \langle \boldsymbol{\eta}, \mathbf{x} \circ \mathbf{y} \rangle - \langle \mathbf{a}, \mathbf{y} \rangle - \langle \boldsymbol{\eta} \circ \mathbf{x} - \mathbf{a}, \mathbf{y} \rangle \\ &= \langle \boldsymbol{\eta}, \mathbf{x} \circ \mathbf{y} \rangle - \langle \mathbf{a}, \mathbf{y} \rangle - \langle \boldsymbol{\eta} \circ \mathbf{x}, \mathbf{y} \rangle + \langle \mathbf{a}, \mathbf{y} \rangle = 0.\end{aligned}$$

Soundness follows from the following theorem.

**Theorem 3.2 (Soundness of $\Pi_{\mathsf{IP\text{-}Check}}$).** *For invalid input, i.e., $\exists i \in [m]$ . $x_i \cdot y_i \neq z_i \pmod{2^k}$, the check passes with probability at most* $\mathsf{err}_{\mathsf{IP\text{-}Check}} := 2^{-(s+1) \cdot d_0 \cdot d_1}$.

*Proof.* Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ and $\langle \mathbf{a}, \mathbf{y} \rangle = c + \delta_c$. If the input is invalid, then there is an index $j \in [m]$ such that $\delta_{z,j} \neq 0 \pmod{2^k}$. The protocol accepts if and only if

$$\begin{aligned}0 &= \langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\alpha}, \mathbf{y} \rangle = \langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\eta} \circ \mathbf{x}, \mathbf{y} \rangle + \langle \mathbf{a}, \mathbf{y} \rangle \\ &= \sum_{i \in [m]} \eta_i \cdot (z_i - x_i \cdot y_i) - c + \langle \mathbf{a}, \mathbf{y} \rangle = \sum_{i \in [m]} \eta_i \cdot (-\delta_{z,i}) + \delta_c\end{aligned}$$

With this equality, we can conclude by Lemma 2.2. □

### 3.3 Compressed Multiplication Check

Our third, and final check, is adapted from Limbo [DOT21]. In contrast to the previous checks, we do not use 2-adic extensions here, since we would have to extend the modulus repeatedly at least $\log_\nu(m)$ times. To apply the compressed protocol with compression factor $\nu$, the check must happen over an algebraic structure where an exceptional sequence of length at least $2\nu+1$ exists.

We first give the subprotocol of [DOT21] to compress a sequence of $\nu$ inner product tuples into a single inner product tuple in Figure 4; then we present the main protocol in Figure 5. Correctness and zero-knowledge for this checking protocol follow the same arguments as the original version over fields. Soundness follows from the following theorem.

**Theorem 3.3 (Soundness of $\Pi_{\mathsf{Comp\text{-}Check}}$).** *Let $d := d_0 \cdot d_1$. For invalid input, i.e., $\exists i \in [m]$ . $x_i \cdot y_i \neq z_i \pmod{2^k}$, the check passes with probability at most*

$$\begin{aligned}\mathsf{err}_{\mathsf{Comp\text{-}Check}} :=& 2^{-d} + (1 - 2^{-d}) \cdot \left( \left( \frac{2(\nu-1)}{2^d - \nu} \right) \cdot \sum_{j=0}^{\log_\nu(m)-2} \left( 1 - \frac{2(\nu-1)}{2^d-\nu} \right)^j \right. \\ &\left. + \left( \frac{2\nu}{2^d - \nu} \right) \cdot \left( 1 - \frac{2(\nu-1)}{2^d - \nu} \right)^{\log_\nu(m)-1} \right) \leq 2^{-d} + \frac{2\nu}{2^d - \nu} \cdot \log_\nu(m).\end{aligned}$$

**$\Pi_{\mathsf{Compress}}$ Subroutine for Inner Product Compression**

**Parameters:** compression factor $\nu$, dimension $\ell$, flag $\mathsf{rand} \in \{\top, \bot\}$
**Inputs:** $\nu$ shared dimension-$\ell$ inner product tuples $([\![\mathbf{x}_i]\!], [\![\mathbf{y}_i]\!], [\![z_i]\!])_{i\in[\nu]}$ shared over $GR(2^k, d)$
**Outputs:** one shared dimension-$\ell$ inner product tuple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![z]\!])$ shared over $GR(2^k, d)$
**Protocol:**
Let $\{\alpha_1, \ldots, \alpha_{2\nu+1}\} \subset \mathsf{Ex}(GR(2^k, d))$.

1. If $\mathsf{rand} = \bot$ define two shared dimension-$\ell$ vectors of degree-$(\nu - 1)$ polynomials $[\![\mathbf{f}]\!], [\![\mathbf{g}]\!]$:

$$\mathbf{f}(\alpha_i) = \begin{pmatrix}\mathbf{x}_1 \cdots \mathbf{x}_\nu\end{pmatrix}^T$$
$$\mathbf{g}(\alpha_i) = \begin{pmatrix}\mathbf{y}_1 \cdots \mathbf{y}_\nu\end{pmatrix}^T$$

   where $i \in [\nu]$. Note, the parties can compute the shared coefficients $[\![f_j]\!], [\![g_j]\!]$ locally from the $[\![\mathbf{x}_i]\!], [\![\mathbf{y}_i]\!]$ by Lagrange interpolation.
   If $\mathsf{rand} = \top$, obtain random shares $[\![\mathbf{v}]\!], [\![\mathbf{w}]\!] \leftarrow \mathcal{O}_H$ and define $\mathbf{f}, \mathbf{g}$ instead of degree $\nu$ with the additional points $\mathbf{f}(\alpha_{\nu+1}) = \mathbf{v}$ and $\mathbf{g}(\alpha_{\nu+1}) = \mathbf{w}$.
2. Inject $[\![z_i]\!] \leftarrow \mathcal{O}_H$ for $i \in [\nu + 1, 2\nu - 1]$ such that $z_i := \langle \mathbf{f}(\alpha_i), \mathbf{g}(\alpha_i)\rangle$.
   If $\mathsf{rand} = \top$, similarly inject $[\![z_i]\!]$ for $i \in \{2\nu, 2\nu + 1\}$.
3. If $\mathsf{rand} = \bot$ define shared polynomial $[\![h]\!]$ of degree $2(\nu - 1)$ by $h(\alpha_i) = z_i$ for $i \in [\nu, 2\nu - 1]$. Again, the parties can compute the shared coefficients $[\![h_j]\!]$ locally from the $[\![z_i]\!]$ by Lagrange interpolation.
   If $\mathsf{rand} = \top$, instead define $h$ of degree $2\nu$ with the additional points $h(\alpha_i) = z_i$ for $i \in \{2\nu, 2\nu + 1\}$.
4. Obtain challenge $\varepsilon \leftarrow \mathcal{O}_R$ such that $\varepsilon \in \mathsf{Ex}(GR(2^k, d)) \setminus \{\alpha_i\}_{i\in[\nu]}$.
5. Output $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![z]\!]) := ([\![\mathbf{f}(\varepsilon)]\!], [\![\mathbf{g}(\varepsilon)]\!], [\![h(\varepsilon)]\!])$.

**Fig. 4.** The subroutine for inner product compression

**$\Pi_{\mathsf{Comp\text{-}Check}}$: Compressed Multiplication Check**

**Parameters:** number of multiplications $m$, compression factor $\nu$ (assume $\log_\nu(m) \in \mathbb{N}$), Galois extension degree $d_1$
**Inputs:** $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ of length $m$ shared over $GR(2^k, d_0)$.
**Protocol:**

1. Lift $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ to $GR(2^k, d_0 \cdot d_1)$.
2. Create inner product tuple $\big([\![\mathbf{x}^0]\!], [\![\mathbf{y}^0]\!], [\![z^0]\!]\big)$:
   (a) $\boldsymbol{\eta} \leftarrow \mathcal{O}_R$ such that $\boldsymbol{\eta} \in GR(2, d_0 \cdot d_1)^m$.
   (b) Set $[\![\mathbf{x}^0]\!] := \boldsymbol{\eta} \circ [\![\mathbf{x}]\!]$, $[\![\mathbf{y}^0]\!] := [\![\mathbf{y}]\!]$, and $[\![z^0]\!] := \langle \boldsymbol{\eta}, [\![\mathbf{z}]\!]\rangle$.
3. For each round $j \in [\log_\nu(m)]$:
   (a) Parse $\big([\![\mathbf{x}^{j-1}]\!], [\![\mathbf{y}^{j-1}]\!], [\![z^{j-1}]\!]\big)$ (of length $m/\nu^{j-1}$) as

$$[\![\mathbf{x}^{j-1}]\!] = \big([\![\mathbf{a}_1^j]\!], \ldots, [\![\mathbf{a}_\nu^j]\!]\big)$$
$$[\![\mathbf{y}^{j-1}]\!] = \big([\![\mathbf{b}_1^j]\!], \ldots, [\![\mathbf{b}_\nu^j]\!]\big)$$

   where the $\mathbf{a}_i^j, \mathbf{b}_i^j$ are of length $m/\nu^j$.
   (b) For $i \in [\nu]$, obtain $[\![c_i^j]\!] \leftarrow \mathcal{O}_H$ such that $c_i^j = \langle \mathbf{a}_i^j, \mathbf{b}_i^j\rangle$.
   (c) If $j < \log_\nu(m)$, run

$$\big([\![\mathbf{x}^j]\!], [\![\mathbf{y}^j]\!], [\![z^j]\!]\big) \leftarrow \Pi_{\mathsf{Compress}}\big(([\![\mathbf{a}_i^j]\!], [\![\mathbf{b}_i^j]\!], [\![c_i^j]\!])_{i\in[\nu]}\big),$$

   else if $j = \log_\nu(m)$, run

$$\big([\![\mathbf{x}^j]\!], [\![\mathbf{y}^j]\!], [\![z^j]\!]\big) \leftarrow \Pi^{\mathsf{Rand}}_{\mathsf{Compress}}\big(([\![\mathbf{a}_i^j]\!], [\![\mathbf{b}_i^j]\!], [\![c_i^j]\!])_{i\in[\nu]}\big).$$

   Both yield inner product tuples of length $m/\nu^j$.
4. Open $\mathbf{x}^{\log_\nu(m)} \leftarrow \mathsf{Rec}([\![\mathbf{x}^{\log_\nu(m)}]\!])$.
5. Output $\Pi_{\mathsf{Zero\text{-}Check}}(\mathbf{x}^{\log_\nu(m)} \cdot [\![\mathbf{y}^{\log_\nu(m)}]\!] - [\![z^{\log_\nu(m)}]\!])$.

**Fig. 5.** The compressed multiplication check

*Proof.* We follow the corresponding proof by [DOT21] and define a sequence of events given that the input is invalid:

- Let $A$ be the event that the protocol outputs $\mathsf{Accept}$.
- Let $A_1$ be the event that the tuple $(\llbracket \mathbf{x}^0 \rrbracket, \llbracket \mathbf{y}^0 \rrbracket, \llbracket z^0 \rrbracket)$ obtained through the $\mathsf{ConstructIP}$ subprotocol is correct.
- Let $A_2^j$ for $j \in [\log_\nu(m)]$ be the event that the tuple $(\llbracket \mathbf{x}^j \rrbracket, \llbracket \mathbf{y}^j \rrbracket, \llbracket z^j \rrbracket)$ obtained through the $\mathsf{Compress}$ subprotocol is correct, and write $A_2^0 = A_1$.

We relate the probabilities as follows:

$$\begin{aligned}
\Pr[A] &= \Pr[A_1] + \Pr[\neg A_1] \cdot \Pr[A \mid \neg A_1] \\
\Pr[A \mid \neg A_2^j] &= \Pr[A_2^{j+1}] + \Pr[\neg A_2^{j+1}] \cdot \Pr[A \mid \neg A_2^{j+1}] \quad \text{for } j \in [1, \log_\nu(m) - 1] \\
\Pr[A \mid \neg A_2^{\log_\nu(m)}] &= 0
\end{aligned}$$

We get from Lemmas 3.2 and 3.1 (see below), that

$$\begin{aligned}
\Pr[A_1] &\overset{L.\ 3.2}{=} 2^{-d} \\
\Pr[A_2^j] &\overset{L.\ 3.1}{=} \frac{2(\nu - 1)}{2^d - \nu} \qquad \text{for } j \in [1, \log_\nu(m) - 1] \\
\Pr[A_2^{\log_\nu(m)}] &\overset{L.\ 3.1}{=} \frac{2\nu}{2^d - \nu}
\end{aligned}$$

Combining them (and using $A_1 = A_2^0$ yields

$$\begin{aligned}
\Pr[A] &= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A \mid \neg A_2^0] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot (\Pr[A_2^1] + \Pr[\neg A_2^1] \cdot \Pr[A \mid \neg A_2^1]) \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A_2^1] + \Pr[\neg A_2^0] \cdot \Pr[\neg A_2^1] \cdot \Pr[A \mid \neg A_2^1] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A_2^1] \\
&\quad + \Pr[\neg A_2^0] \cdot \Pr[\neg A_2^1] \cdot (\Pr[A_2^2] + \Pr[\neg A_2^2] \cdot \Pr[A \mid \neg A_2^2]) \\
&= \cdots \\
&= \sum_{j=0}^{\log_\nu(m)} \Pr[A_2^j] \cdot \prod_{i=0}^{j-1} \Pr[\neg A_2^i] + \Pr[A \mid \neg A_2^{\log_\nu(m)}] \cdot \prod_{j=0}^{\log_\nu(m)} \Pr[\neg A_2^j] \\
&= \sum_{j=0}^{\log_\nu(m)} \Pr[A_2^j] \cdot \prod_{i=0}^{j-1} \Pr[\neg A_2^i] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Bigg( \sum_{j=1}^{\log_\nu(m)-1} \Pr[A_2^j] \cdot \prod_{i=1}^{j-1} \Pr[\neg A_2^i] \\
&\qquad\qquad + \Pr[A_2^{\log_\nu(m)}] \cdot \prod_{i=1}^{\log_\nu(m)-1} \Pr[\neg A_2^i] \Bigg) \\
&= 2^{-d} + (1 - 2^{-d}) \cdot \Bigg( \left( \frac{2(\nu - 1)}{2^d - \nu} \right) \cdot \sum_{j=0}^{\log_\nu(m)-2} \left( 1 - \frac{2(\nu - 1)}{2^d - \nu} \right)^j \\
&\qquad\qquad + \left( \frac{2\nu}{2^d - \nu} \right) \cdot \left( 1 - \frac{2(\nu - 1)}{2^d - \nu} \right)^{\log_\nu(m)-1} \Bigg)
\end{aligned}$$

$$\leq 2^{-d} + \frac{2\nu}{2^d - \nu} \cdot \log_\nu(m),$$

which concludes this proof. $\square$

**Lemma 3.1 (Soundness of $\Pi_{\mathsf{Compress}}$).** *If one of the inner product tuples*

$$([\![\mathbf{x}_i]\!], [\![\mathbf{y}_i]\!], [\![z_i]\!])_{i \in [\nu]}$$

*is incorrect, or any of the values $z_i$, $i \in [\nu+1, 2\nu-1]$, is defined incorrectly, then the output inner tuple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![z]\!])$ is also incorrect, except with probability at most $\frac{2(\nu-1)}{2^d-\nu}$ if $\mathsf{Rand} = \bot$ and $\frac{2\nu}{2^d-\nu}$ if $\mathsf{Rand} = \top$.*

*Proof.* For now assume that $\mathsf{Rand} = \bot$. Suppose there is an error at index $j \in [2\nu - 1]$. Then we have $z_j \neq \langle \mathbf{f}(\alpha_j), \mathbf{g}(\alpha_j) \rangle$, where $z_j$ is either part of the input ($j \in [\nu]$) or an injected value ($j \in [\nu+1, 2\nu-1]$). In both cases, we have $h(\alpha_j) \neq \langle \mathbf{f}(\alpha_j), \mathbf{g}(\alpha_j) \rangle$, and therefore $h \neq \langle \mathbf{f}, \mathbf{g} \rangle$.

We now apply the generalized Schwartz-Zippel Lemma (Lemma 2.1). Note that the challenge $\varepsilon$ is sampled from the exeptional sequence $\mathsf{Ex}(GR(2^k, d)) \setminus \{\alpha_i\}_{i \in [\nu]}$ of size $2^d - \nu$. Hence, we obtain that $\langle \mathbf{x}, \mathbf{y} \rangle \neq z$ iff $\langle \mathbf{f}, \mathbf{g} \rangle (\varepsilon) \neq h(\varepsilon)$ with probability at most $\frac{2(\nu-1)}{2^d-\nu}$.

In the case $\mathsf{Rand} = \top$, we analogously obtain an error probability of at most $\frac{2\nu}{2^d-\nu}$. $\square$

**Lemma 3.2.** *For invalid input $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ into $\Pi_{\mathsf{Comp\text{-}Check}}$, i.e., such that $\mathbf{x} \circ \mathbf{y} \neq \mathbf{z}$, we have $\langle \mathbf{x}^0, \mathbf{y}^0 \rangle \neq z^0$ except with probability $2^{-d_0 \cdot d_1}$.*

*Proof.* Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ Let $j \in [m]$ such that $x_j \cdot y_j \neq z_j$ and, hence, $\delta_{z,j} \neq 0$. Then

$$\begin{aligned} &\langle \mathbf{x}^0, \mathbf{y}^0 \rangle = z^0 \\ \iff &\sum_{i \in [m]} \eta_i \cdot x_i \cdot y_i = \sum_{i \in [m]} \eta_i \cdot z_i \end{aligned}$$

Hence we can apply Lemma 2.2 $\square$

## 4 Checking Base Ring Sharings

To ensure the prover knows and inputs a witness over the base ring $\mathbb{Z}_{2^k}$, we devise a check for the parties to ensure this in Figure 6. We can perform a batched check that all the values we wish to inspect are simultaneously correct by taking a random linear combination with coefficients from $\mathbb{Z}_{2^{1+s_{\mathsf{rc}}}}$, and opening that. Since this would leak a linear combination of secret values, we also allow the prover to input an additional sharing of a value in $\mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$ to mask this relation (before receiving the random coefficients from the verifier). This is conceptually similar to the recent approach by Shoup and Smart in [SS23].

In [ACD+19], Abspoel et al. consider a similar problem for the case of non-MPCitH MPC protocols. They solve this problem by generating random secret shared masks hiding values in the correct ring by means of hyperinvertible matrices, after which these masks can be adjusted with a public value to hide the wanted secret. In an MPCitH context however, this becomes both less convenient, since all computing parties need to contribute their own randomness, as well as requiring a higher communication cost in the final proof size. Soundness follows from the following theorem.

**Theorem 4.1 (Soundness of $\Pi_{\mathsf{Ring\text{-}Check}}$).** *For invalid input, that is if any of $x_0, x_1, \ldots, x_\ell$ are a value in $GR(2^k, d_0) \setminus \mathbb{Z}_{2^k}$ when reduced modulo $2^k$, the check passes with probability at most $\mathsf{err}_{\mathsf{Ring\text{-}Check}} := 2^{-(s_{\mathsf{rc}}+1)}$.*

$\Pi_{\textsf{Ring-Check}}$

**Inputs:** $[\![\mathbf{x}]\!] = ([\![x_1]\!], \ldots, [\![x_\ell]\!])$ shared over $GR(2^{k+s_{\mathsf{rc}}}, d_0)$

**Protocol:**

1. Obtain $[\![x_0]\!]$, corresponding to a value in the ring $\mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$ from $\mathcal{O}_H$.
2. Receive $\ell$ random coefficients $r_1, \ldots, r_\ell \in \mathbb{Z}_{2^{1+s_{\mathsf{rc}}}}$ from $\mathcal{O}_R$.
3. Compute and open $[\![v]\!] = [\![x_0]\!] + r_1[\![x_1]\!] + \ldots + r_\ell[\![x_\ell]\!]$.
4. If $v \in \mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$, return $\top$, otherwise return $\perp$.

**Fig. 6.** The check to ensure sharings correspond to values in the base ring.

**Table 1.** Rings and numbers of primitive operations used by the three multiplication checking protocols.

| | Multiplication Check | | |
|---|---|---|---|
| | $\Pi_{\textsf{Sac-Check}}$ | $\Pi_{\textsf{IP-Check}}$ | $\Pi_{\textsf{Comp-Check}}$ |
| small ring $\mathcal{R}_{\textsf{small}}$ | $GR(2^{k+s}, d_0)$ | $GR(2^{k+s}, d_0)$ | $GR(2^k, d_0)$ |
| big ring $\mathcal{R}_{\textsf{large}}$ | $GR(2^{k+s}, d_0 \cdot d_1)$ | $GR(2^{k+s}, d_0 \cdot d_1)$ | $GR(2^k, d_0 \cdot d_1)$ |
| challenge space $\mathcal{C}$ | $GR(2^{1+s}, d_0 \cdot d_1)$ | $GR(2^{1+s}, d_0 \cdot d_1)$ | $GR(2, d_0 \cdot d_1)$ |
| rounds $\mu$ | 1 | 1 | $\log_\nu(m) + 1$ |
| input over $\mathcal{R}_{\textsf{small}}$ | $\#\textsf{inputs} + m$ | $\#\textsf{inputs} + m$ | $\#\textsf{inputs} + m$ |
| hint over $\mathcal{R}_{\textsf{large}}$ | $m$ | 1 | $(2\nu - 1) \cdot \log_\nu(m) + 2$ |
| uniform hint over $\mathcal{R}_{\textsf{large}}$ | $m$ | $m$ | 2 |
| reconstruction over $\mathcal{R}_{\textsf{large}}$ | $m$ | $m$ | 1 |
| challenge from $\mathcal{C}$ | 1 | $m$ | $m + \log_\nu(m)$ |

*Proof.* This is simply Lemma 2.2, applied to only a single coefficient of the Galois extension. Hence we get a bound of $2^{-(s_{\mathsf{rc}}+1)\cdot 1}$. $\square$

When dealing with additive sharings, the parties can instead simply check their own local shares to lie in the correct ring and return $\perp$ when this is not the case. For semi-honest parties, this is guaranteed to have no false positives.

## 5 Protocol Communication Costs

The communication costs of the zero-knowledge proofs depends greatly on the used secret sharing scheme and the multiplication check protocol, as well as a large set of parameters. To simplify notation, we use $\mathcal{R}_{\textsf{small}}$ for the ring used to share the witness, $\mathcal{R}_{\textsf{large}}$ for the ring extension in which the checks are performed. Moreover, the random challenges from $\mathcal{O}_R$ live in the challenge space $\mathcal{C}$, and $\mu$ denotes the number of rounds of the MPC protocol, i.e., the number of calls to $\mathcal{O}_R$. For brevity of notation, we use $\mathcal{B}(S) = \lceil \log_2 |S| \rceil$ to denote the number of bits needed to represent an element from $S$.

Table 1 shows how many primitive operations we need for each checking protocol, and Table 2 gives the communication cost of each operation in both sharing types. The costs of the challenges are $\mathcal{B}(\mathcal{C}) \cdot \mu \cdot \tau_{\textsf{in}}$, since they can be shared across the "outer repetitions".

**Table 2.** Communication costs in bits of the primitive operations. Here $\mathcal{B}(\cdot)$ denotes the number of bits required to encode an element of the set passed as argument.

| | Sharing Scheme | |
|---|---|---|
| | Additive | Threshold |
| input over $\mathcal{R}_{\mathsf{small}}$ | $\mathcal{B}(\mathcal{R}_{\mathsf{small}})$ | $\mathcal{B}(\mathcal{R}_{\mathsf{small}}) \cdot t$ |
| hint over $\mathcal{R}_{\mathsf{large}}$ | $\mathcal{B}(\mathcal{R}_{\mathsf{large}})$ | $\mathcal{B}(\mathcal{R}_{\mathsf{large}}) \cdot t$ |
| uniform hint over $\mathcal{R}_{\mathsf{large}}$ | 0 | $\mathcal{B}(\mathcal{R}_{\mathsf{large}}) \cdot t$ |
| reconstruction over $\mathcal{R}_{\mathsf{large}}$ | $\mathcal{B}(\mathcal{R}_{\mathsf{large}})$ | $\mathcal{B}(\mathcal{R}_{\mathsf{large}})$ |
| challenge from $\mathcal{C}$ | $\mathcal{B}(\mathcal{C})$ | $\mathcal{B}(\mathcal{C})$ |

### 5.1 Primitive Costs

The communication costs for our basic operations can be summarized as follows.

*Commitments:* Before each call to $\mathcal{O}_R$ the prover commits to the current state of the computation. The $\tau_{\mathsf{out}} \cdot \mu \cdot N$ total commitments can be combined into $\tau_{\mathsf{out}} \cdot \mu$ Merkle trees, and for each round it is sufficient to send a hash of the $\tau_{\mathsf{out}}$ Merkle roots. Thus, committing costs $2\lambda \cdot \mu$ bits. Before the verifier selects a subset of parties whose views to open, the prover sends another hash with shares of the last reconstructed values.

To open $t$ of the commitments in each repetition, we have to send, in addition to the committed data, $\lambda$ bits of randomness per commitment as well the corresponding Merkle paths. Each path is of length $\log_2(N)$, but since we open $t$ views and the path overlap, we pay $2\lambda \cdot \log_2(N/t)$ bits per path.

Overall, this results in

$$\mathsf{size}_{\mathsf{Commit}} := 2\lambda \cdot (\mu + 1) + \tau_{\mathsf{out}} \cdot \lambda \cdot \mu \cdot t \cdot (2\log_2(N/t) + 1)$$

bits of communication for committing and opening.

*Opening sharings:* Since to open a sharing only the reconstructed value needs to be revealed on top of the $t$ already decommited shares, the cost for opening a $\mathbb{Z}_{2^k}$ value is $k$ bits (for a $GR(2^k, d)$ value this is $k \cdot d$ bits), regardless of the secret sharing scheme being used.

*Providing hints:* The $\mathcal{O}_H$ oracle can be instantiated in two different ways, depending on the kind of secret sharing being used. For a threshold secret sharing scheme, both specific and uniformly random values $v \in \mathbb{Z}_{2^k}$ (or $v \in GR(2^k, d)$) can be obtained by running $[\![v]\!] \leftarrow \mathsf{Share}(v)$ and distributing the shares to the corresponding parties. This costs $t \cdot k$ (or $t \cdot k \cdot d$) bits of proof size.

For additive secret sharing, uniformly random values in $\mathbb{Z}_{2^k}$ or $GR(2^k, d)$ can be obtained at zero extra cost by having all parties individually derive their shares from a PRG seed. A uniformly random sharing $[\![r]\!]^A$ can be transformed into a sharing of a specific value $[\![v]\!]^A$ by updating the public adjustment $\Delta_v$, at the cost of only $k$ or $k \cdot d$ bits of proof size.

### 5.2 Protocol Costs

We can now summarize the communication costs per checking protocol:

$\Pi_{\mathsf{Sac\text{-}Check}}$**:** The sacrificing check requires

$$\mathsf{size}^A_{\mathsf{Sac\text{-}Check}} := 2 \cdot m \cdot (k + s) \cdot d_0 \cdot d_1$$
$$\mathsf{size}^T_{\mathsf{Sac\text{-}Check}} := (2 \cdot m \cdot t + m) \cdot (k + s) \cdot d_0 \cdot d_1$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\textsf{IP-Check}}$: The inner product check results requires

$$\begin{aligned}\mathsf{size}^A_{\textsf{IP-Check}} &:= (m+1)\cdot(k+s)\cdot d_0\cdot d_1\\ \mathsf{size}^T_{\textsf{IP-Check}} &:= ((m+1)\cdot t+m)\cdot(k+s)\cdot d_0\cdot d_1\end{aligned}$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\textsf{Comp-Check}}$: The compressed multiplication check results requires

$$\begin{aligned}\mathsf{size}^A_{\textsf{Comp-Check}} &:= ((2\nu-1)\cdot\log_\nu(m)+3)\cdot k\cdot d_0\cdot d_1\\ \mathsf{size}^T_{\textsf{Comp-Check}} &:= (((2\nu-1)\cdot\log_\nu(m)+4)\cdot t+1)\cdot k\cdot d_0\cdot d_1\end{aligned}$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\textsf{Ring-Check}}$: For additive sharing, this check has no overhead. In the threshold case, this procedure requires one additional share input and one share reconstruction in $GR(2^{k+s_{\textsf{rc}}}, d_0)$ to the overall proof size, hence the total costs are

$$\begin{aligned}\mathsf{size}^A_{\textsf{Ring-Check}} &:= 0\\ \mathsf{size}^T_{\textsf{Ring-Check}} &:= (t+1)\cdot(k+s_{\textsf{rc}})\cdot d_0\end{aligned}$$

bits of communication for additive, resp. threshold, sharing.

Here we do not take into account the cost of the verifier sending a challenge or a seed for outputs of the $\mathcal{O}_R$ oracle. In the non-interactive case, these are obtained from the Fiat–Shamir transform and therefore free in terms of communication; in the interactive case however, the verifier sends $\lambda$ bits per "round" of dependent calls to $\mathcal{O}_R$.

### 5.3 Overall Costs

Finally, we can present the overall communication cost, i.e., the proof size. Note here that the cost for $\mathsf{size}_{\textsf{Input}}$ depends on $k+s_{\textsf{rc}}$, rather than the potentially smaller $k+s$.

$$\mathsf{size}_{\textsf{Proof}} = \mathsf{size}_{\textsf{Commit}} + \tau_{\textsf{out}}\cdot(\mathsf{size}_{\textsf{Input}} + \tau_{\textsf{in}}\cdot\mathsf{size}_{\textsf{Check}}) + \tau_{\textsf{in}}\cdot\mathsf{size}_{\textsf{Challenge}}$$

### 5.4 Concrete Comparison of the Three $\Pi_{\textsf{Mult-Check}}$ Subprotocols

To compare our different protocols concretely with one another, we fix certain choices for $\sigma$, $k$ and $m$ and examined the per-multiplication-gate communication cost of a full proof $\sigma$ bits of security. The size presented in the tables corresponds to the communication cost of an entire proof, except for the challenges sent from the verifier. That is, we only examine the communication from the prover towards the verifier, which also gives a good idea of the proof size that would be incurred when the protocol is transformed to a non-interactive proof by the Fiat-Shamir transform.

All our experimental validations were computed with $\#\textsf{inputs} = 128$ elements in $\mathbb{Z}_{2^k}$. Since the additive sharing has some optimizations for random sharings and $\Pi_{\textsf{Ring-Check}}$ and does not require $d_0 > 1$ to enable sharing values across $N$ parties, it generally comes out as the optimal choice for the configurations examined here.

When combining our protocols with the packing techniques of Section 6, the balance shifts since a threshold $t < N-1$ gives better soundness per parallel repetition, allows for more packing, and compensates for the larger $d_0$ by performing more parallel proofs. Out of interest for this trade-off, we present the parameter sets and associated costs for additive and threshold secret sharing separately.

We observe that for $\Pi_{\mathsf{Sac\text{-}Check}}$ and $\Pi_{\mathsf{IP\text{-}Check}}$, which require at least $m$ openings each, the optimal choice for $d_1$ is one since the overhead for $d_0 \cdot s$ extra bits is generally smaller than $d_0 \cdot (d_1 - 1) \cdot k$ extra bits, even though the size of inputs and injected multiplications grows as well. When the communication due to the check is asymptotically smaller than the communication due to the input of the extended witness, it becomes preferable to avoid the extra $d_0 \cdot s$ bits per multiplication cost in the input already.

**Table 3.** Cost comparison for $\sigma = 40$, $m = 1024$ with threshold secret sharing.

| $k$ | Protocol | $N$ | $t$ | $d_0$ | $d_1$ | $s$ | $s_{\mathsf{rc}}$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 63 | 1 | 6 | 1 | 2 | 17 | / | 1 | 7 | 748 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 539 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 18 | 4 | 1 | 7 | 236 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 1 413 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 1 012 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 18 | 4 | 1 | 7 | 452 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 5 399 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 3 846 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 18 | 2 | 1 | 7 | 1 726 |

**Table 4.** Cost comparison for $\sigma = 40$, $m = 1024$ with additive secret sharing.

| $k$ | Protocol | $N$ | $d_0$ | $d_1$ | $s$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 116 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 63 | 1 | 1 | 8 | / | 1 | 7 | 82 |
| | $\Pi_{\mathsf{Compress}}$ | 15 | 1 | 12 | / | 4 | 1 | 11 | 87 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 191 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 137 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 14 | / | 4 | 1 | 7 | 135 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 641 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 443 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 14 | / | 4 | 1 | 7 | 411 |

**Table 5.** Cost comparison for $\sigma = 40$, $m = 32768$ with threshold secret sharing.

| $k$ | Protocol | $N$ | $t$ | $d_0$ | $d_1$ | $s$ | $s_{\mathsf{rc}}$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 22 449 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 15 729 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 4 | 1 | 7 | 5 459 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 42 953 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 30 090 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 4 | 1 | 7 | 10 895 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 165 979 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 3 | 31 | / | 1 | 2 | 116 252 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 2 | 1 | 7 | 43 476 |

Since we can observe that $\Pi_{\mathsf{Compress}}$ consistently results in the smallest proof sizes, we further also look at the overhead of this protocol. That is, we investigate the ratio of proof size to the theoretical optimum of $k \cdot (\#\mathsf{inputs} + m)$ bits for any protocol that needs to inject the results of multiplications. This rate is a constant that mostly depends on the target value of $\sigma$ and

**Table 6.** Cost comparison for $\sigma = 40$, $m = 32768$ with additive secret sharing.

| $k$ | Protocol | $N$ | $d_0$ | $d_1$ | $s$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 2 836 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 1 900 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 16 | / | 8 | 1 | 6 | 945 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 5 143 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 3 439 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 16 | / | 8 | 1 | 6 | 1 745 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 18 985 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 7 | / | 1 | 6 | 12 673 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 14 | / | 4 | 1 | 6 | 6 531 |

**Table 7.** Cost comparison for $\sigma = 128$, $m = 32768$ with threshold secret sharing.

| $k$ | Protocol | $N$ | $t$ | $d_0$ | $d_1$ | $s$ | $s_{\mathsf{rc}}$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 2 | 8 | 1 | 2 | 23 | / | 1 | 9 | 68 673 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 4 | 39 | / | 1 | 6 | 48 550 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 4 | 1 | 22 | 17 158 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 4 | 39 | / | 1 | 6 | 130 798 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 4 | 39 | / | 1 | 6 | 91 631 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 4 | 1 | 22 | 34 239 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 3 | 8 | 1 | 4 | 39 | / | 1 | 6 | 499 875 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 3 | 8 | 1 | 4 | 39 | / | 1 | 6 | 350 119 |
| | $\Pi_{\mathsf{Compress}}$ | 63 | 1 | 6 | 4 | / | 17 | 2 | 1 | 22 | 136 637 |

decreases slightly as the number of multiplications increases. Since the choice of $k$ doesn't influence the choice of multiplication check, it also has no further impact on the overhead.

**Table 8.** Cost comparison for $\sigma = 128$, $m = 32768$ with additive secret sharing.

| $k$ | Protocol | $N$ | $d_0$ | $d_1$ | $s$ | $\nu$ | $\tau_{\mathsf{in}}$ | $\tau_{\mathsf{out}}$ | Proof size in kB |
|---|---|---|---|---|---|---|---|---|---|
| 32 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 8 443 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 5 655 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 16 | / | 8 | 1 | 17 | 2 677 |
| 64 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 14 980 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 10 016 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 16 | / | 8 | 1 | 17 | 4 944 |
| 256 | $\Pi_{\mathsf{Sac\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 54 199 |
| | $\Pi_{\mathsf{IP\text{-}Check}}$ | 255 | 1 | 1 | 9 | / | 1 | 17 | 36 179 |
| | $\Pi_{\mathsf{Compress}}$ | 255 | 1 | 16 | / | 8 | 1 | 17 | 18 549 |

## 6 Packing

In this section, we present two orthogonal ways in which our protocols can be extended to provide SIMD-style packing for parallel proofs of multiple independent statements. We then discuss how this packing can be applied to achieve parallelization of proofs for structured circuits.

### 6.1 Packing in the Shamir Domain

The most common way to achieve packing, when using Shamir secret sharing, is to hide multiple secrets in the same polynomial by ensuring the sharing polynomial $p$ evaluates to $p(\alpha_0) = v_0, p(\alpha_1) = v_1, \ldots, p(\alpha_{\ell-1}) = v_{\ell-1}$ when sharing $\ell$ values, for $\alpha_0, \ldots, \alpha_{\ell+N-1} \in \mathsf{Ex}(\mathcal{R})$. Of course, the shares for the parties should then be evaluations at $\alpha_\ell, \ldots, \alpha_{\ell+N-1}$ in order to preserve privacy.

The degree of $p$ now must become $t+\ell-1$ to ensure that $t$ parties still learn nothing (including algebraic relations between values) about the shared secrets. This implies that opening a shared value now requires $t + \ell$ shares, rather than the regular $t + 1$. In the context of our protocols however, this does not mean we need to open more commitments towards the verifier since either the opened value is assumed to be known (in the case of $\Pi_{\mathsf{Zero\text{-}Check}}$) or provided as part of the proof (in the case of a normal reconstruction). In both of these cases, the additional knowledge effectively acts as $\ell$ additional known shares at the evaluation points $\alpha_0, \ldots, \alpha_{\ell-1}$.

Applying this technique to our protocols then allows us to prove $\ell$ separate witnesses for an identical circuit in parallel. The impact on the communication cost is twofold: $\mathsf{Ex}(GR(2^k, d_0))$ should be large enough to allow for $t + \ell$ points and hence $2^{d_0} \geq t + \ell$, and any reconstruction must provide $\ell$ reconstructed values as part of the proof. Importantly however, the size of sharing of the (extended) witness does not grow, resulting in an approach that is cheaper than performing $\ell$ separate proofs independently.

### 6.2 Packing in the Galois Domain

Our second approach to packing makes use of the "extra space" that is found in a $GR(2^k, d_0)$ element. Rather than having to send $k \cdot d_0$ bits to represent a single $k$-bit value, we can send $d_0$ such values, each in its own coefficient of the Galois ring element, considering it more as a $\mathbb{Z}_{2^k}$-module of dimension $d_0$.

As long as any operation the parties perform on their shares is an operation for this module (so addition and scalar multiplication by scalars in $\mathbb{Z}_{2^k}$), the actions of the secret sharing and reconstruction are not further impeded. Losing the ability to perform scalar multiplication with values from the entire space $GR(2^k, d_0)$ incurs some cost on the soundness of $\Pi_{\mathsf{Sac\text{-}Check}}$ and $\Pi_{\mathsf{IP\text{-}Check}}$, where the verifier's random coefficients can now only come from $\mathbb{Z}_{2^k}$ instead, leading to a soundness error of $2^{-(s+1)\cdot d_1}$ rather than $2^{-(s+1)\cdot d_1 \cdot d_0}$.

If $\Pi_{\mathsf{Compress}}$ is used, then it is necessary to deal with the polynomial interpolation needed in $\Pi_{\mathsf{Compress}}$, which requires *some* scalar multiplication with values coming from an exceptional set of at least size $2 \cdot \nu$. To handle this case, we suggest two possible approaches.

**Reducing module dimension.** The first approach plays with the same concept described before. It uses the additional free space available in $GR(2^k, d_0)$, but rather than seeing it as a $\mathbb{Z}_{2^k}$-module, it treats it as a $GR(2^k, d_{\mathsf{interp}})$-module of dimension $\frac{d_0}{d_{\mathsf{interp}}}$, subject to $2^{d_{\mathsf{interp}}} \geq 2 \cdot \nu$ to allow for the interpolation.

**Tweak the lifting.** In the second approach, we tweak the "local lifting" from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$. Rather than treating the larger ring as a degree $d_1$ extension of the smaller one, we can choose $d_1$ such that $\gcd(d_0, d_1) = 1$, and construct the larger ring as a degree $d_0$

extension of $GR(2^k, d_1)$, even though the input lies in $GR(2^k, d_0)$. To see why this works, we can consider $GR(2^k, d_0 \cdot d_1) = \mathbb{Z}_{2^k}[\beta, \gamma]$, where $GR(2^k, d_0) = \mathbb{Z}_{2^k}[\beta]$ and $GR(2^k, d_1) = \mathbb{Z}_{2^k}[\gamma]$. Due to our restriction that $\gcd(d_0, d_1) = 1$, $\beta$ and $\gamma$ are algebraically independent, allowing us to reinterpret $\mathbb{Z}_{2^k}[\beta, \gamma] = (\mathbb{Z}_{2^k}[\beta])[\gamma] = (\mathbb{Z}_{2^k}[\gamma])[\beta]$. In the interpretation $(\mathbb{Z}_{2^k}[\gamma])[\beta]$, we are now left with a form that allows us to treat these values as a $GR(2^k, d_1)$-module of dimension $d_0$. When doing this, the only further constraint we have is that $2^{d_1} \geq 2 \cdot \nu$, while being able to fully pack all $d_0$ input coefficients.

These approaches incur some loss in soundness, resulting in a cheating probability for the multiplication checks of $2^{-d_1 \cdot \frac{d_0}{d_{\mathsf{interp}}}}$ or $2^{-d_1}$ respectively.

Although the loss in soundness necessitates more communication to return to the same level of security, the reduction in communication when averaged over the parallel proof instances brings some benefits. When performing $d_0$ proofs in parallel, neither the communication for the input of the extended witness, nor the communication to reconstruct a secret-shared value increase. When all coefficients in the input sharing are filled with actual inputs, we also no longer need to perform $\Pi_{\mathsf{Ring\text{-}Check}}$, as all $GR(2^k, d_0)$ elements now correspond to a valid set of $d_0$ elements in $\mathbb{Z}_{2^k}$.

We also considered the use of *Reverse Multiplication-Friendly Embeddings* (RMFEs), as introduced by [CCXY18], for this sort of packing, but since this only provides $\mathbb{Z}_{2^k}$-linearity, it is incompatible with our threshold secret sharing. Additionally, RMFEs only provide a constant packing rate, whereas our technique succeeds in utilising the available space maximally.

### 6.3 Multi-Round Computations

Instead of proving some $\ell$ independent instances of a circuit in parallel, one would often prefer to use this packing to prove a single instance more efficiently, either by performing multiple of the "outer" repetitions in parallel, or by performing multiple gates of the circuit in parallel. As the challenges provided by the verifier are shared across the parallel instances being proved, the former is unfortunately not possible. The latter however, can be achieved by introducing a gadget that checks whether two secret shared values $[\![a]\!]$ and $[\![b]\!]$ are (prescribed) permutations.

Depending on the efficiency of such a check, this could allow optimizing for circuits that are *wide* enough (that is, circuits that perform enough independent multiplications in parallel), to only allowing optimization for circuits that are highly structured. As an example of such structured circuits, one could consider a computation that procedes in several identical rounds, such as a circuit that performs several consecutive RAM accesses like in Section 7. In an ideal scenario, the permutation check can be performed mostly entirely locally, with a final $\Pi_{\mathsf{Zero\text{-}Check}}$ at the end of the protocol, yielding an improvement in communication cost of factor $\ell$ practically for free. For the permutation checks we will describe here, a highly structured/repetitive circuit should be preferred however.

To check the reordering of a Shamir packed secret sharing, each party can *re-share* their share and enable a private reconstruction of the underlying secrets, which can then be re-ordered and eventually checked in batch with a random linear combination and $\Pi_{\mathsf{Zero\text{-}Check}}$. This results in $t \cdot (2 \cdot N - t)$ ring elements of communication to perform the re-sharing. To check the reordering of Galois coefficients in $GR(2^k, d)$, we let the prover inject $d$ sharings of $\mathbb{Z}_{2^k}$ elements $M_i$ (which need to be checked through $\Pi_{\mathsf{Ring\text{-}Check}}$), which can be used to mask corresponding coefficients in $a$ and $b$ identically and provide privacy of the values. Then we can perform a (batched) $\Pi_{\mathsf{Zero\text{-}Check}}(a + \sum_i x^i M_i - b - \sum_i x^{\pi(i)} M_i)$ to validate the permutation. This incurs a cost of $d$ ring elements per permutation check to input the mask values $M_i$.

## 7 RAM Application

In this section, we show how to construct the $C_{\mathsf{check}}$ circuit of [DOTV22] to verify the consistency of a series of $T$ read or write accesses to an initial array $\mathcal{L}$ of size $N$. Our $C_{\mathsf{check}}$ circuit is very similar to that of [DOTV22] albeit with minor modifications to fit our ring structure. In particular, we cannot use the EqCheck sub-circuit that crucially relies on the underlying field structure and we tweak the PermCheck to use the Generalized Schwartz-Zippel (Lemma 2.1). In addition, we assume a large exceptional set. In all the sub-circuits of this section, we overload the notation $[\![.]\!]$ to denote sensitive values that cannot be revealed in the zero-knowledge proof.

First, we introduce the main building blocks, i.e. PermCheck and BdCheck, and later in 7.3, we describe the ring version of $C_{\mathsf{check}}$.

### 7.1 Permutation Check

First, we design a procedure PermCheck, see Figure 7, to verify that two arrays $([\![a_1]\!], \ldots, [\![a_S]\!])$ and $([\![b_1]\!], \ldots, [\![b_S]\!])$ of $S$ shared elements are one a permutation of the other. The idea behind the check is to define two polynomials $P_A(X) = \prod_{i \in [S]}(X - a_i)$ and $P_B(X) = \prod_{i \in [S]}(X - b_i)$ which are identical if and only if both arrays are a permutation of each other, and then use polynomial identity testing to verify this is indeed the case. Both polynomials $P_A$ and $P_B$ are of degree $S$, thus the Generalized Schwartz-Zippel (Lemma 2.1) states that if $A$ is not a permutation of $B$ (i.e. $P_A \neq P_B$), the check passes with probability at most $\frac{S}{2^{d_0 \cdot d_1}}$.

**PermCheck**

**Inputs:** $[\![A]\!] = ([\![a_1]\!], \ldots, [\![a_S]\!])$ and $[\![B]\!] = ([\![b_1]\!], \ldots, [\![b_S]\!])$ both over $GR(2^k, d_0)$
**Protocol:**

1. Lift $[\![a_i]\!]$ and $[\![b_i]\!]$ from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$.
2. $s \leftarrow \mathcal{O}_R$ such that $s \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$.
3. Add the $S-1$ multiplication gates necessary to compute $[\![P_A(s)]\!] = \Pi_{i \in [S]}(s - [\![a_i]\!])$ and similarly for $[\![P_B(s)]\!] = \Pi_{i \in [S]}(s - [\![b_i]\!])$.
4. Add $[\![P_A(s)]\!] - [\![P_B(s)]\!]$ to the list of outputs.

**Fig. 7.** Permutation check

**PermCheck for Tuples**

**Inputs:** $[\![A]\!] = (([\![a_1^{(1)}]\!], [\![a_1^{(2)}]\!], [\![a_1^{(3)}]\!], [\![a_1^{(4)}]\!]) \ldots, ([\![a_S^{(1)}]\!], \ldots, [\![a_S^{(4)}]\!]))$ and
$[\![B]\!] = (([\![b_1^{(1)}]\!], [\![b_1^{(2)}]\!], [\![b_1^{(3)}]\!], [\![b_1^{(4)}]\!]) \ldots, ([\![b_S^{(1)}]\!], \ldots, [\![b_S^{(4)}]\!]))$ both shared over $GR(2^k, d_0)$
**Protocol:**

1. Lift $[\![a_i^{(j)}]\!]$ and $[\![b_i^{(j)}]\!]$ from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$.
2. $r = (r^{(1)}, r^{(2)}, r^{(3)}, r^{(4)}) \leftarrow \mathcal{O}_R$ such that $r^{(j)} \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$
3. For $i \in [S]$ add the linear gates to compute $[\![a_i]\!] = \Sigma_{j \in [4]}[\![a_i^{(j)} \cdot r^{(j)}]\!]$ and $[\![b_i]\!] = \Sigma_{j \in [4]}[\![b_i^{(j)} \cdot r^{(j)}]\!]$
4. $s \leftarrow \mathcal{O}_R$ such that $s \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$.
5. Add the $S-1$ multiplication gates necessary to compute $[\![P_A(s)]\!] = \Pi_{i \in [S]}(s - [\![a_i]\!])$ and similarly for $[\![P_B(s)]\!] = \Pi_{i \in [S]}(s - [\![b_i]\!])$.
6. Add $[\![P_A(s)]\!] - [\![P_B(s)]\!]$ to the list of outputs.

**Fig. 8.** Permutation check for tuples

In addition, we also describe another procedure, given in Figure 8, for when the $a_i$ and $b_i$ are themselves tuples of 4 elements — looking ahead, the array to be checked consists of tuples of 4 elements. This protocol is similar to the previous one, except we first compress our tuple into a single element. Assuming that $A$ and $B$ are not a permutation of each other, then for all permutations $\pi$ there exists at least one tuple $(a_i^{(1)}, a_i^{(2)}, a_i^{(3)}, a_i^{(4)})$ and one tuple $(b_{\pi(i)}^{(1)}, b_{\pi(i)}^{(2)}, b_{\pi(i)}^{(3)}, b_{\pi(i)}^{(4)})$ that differs. The probability that such tuples are compressed into $a_i$ and $b_{\pi(i)}$ respectively such that $a_i = b_{\pi(i)}$ is bounded by the Generalized Schwartz-Zippel lemma for 4-variate polynomial of total degree 4 by $\frac{4}{2^{d_0 \cdot d_1}}$. By union bound, the check thus passes with probability at most $\frac{S+4}{2^{d_0 \cdot d_1}}$.

### 7.2 Bound Check

The bound check $\mathsf{BdCheck}$ is exactly the same as [DOTV22]. For completeness, we recall it in Figure 9. It checks in zero-knowledge that a set of $T$ sensitive values are contained between two public bounds, $B_1, B_1$, with $B_1 < B_2$.

**BdCheck**

**Input:** The lower and upper bounds $B_1 < B_2$.
$[\![\mathcal{L}]\!] = [B_1, B_1 + 1, \ldots, B_2, [\![x_1]\!], \ldots, [\![x_T]\!]]$ of size $S$
$[\![\mathcal{L}']\!]$ that contains the entries of $\mathcal{L}$ sorted from lowest to highest (with all the entries sensitive)
**Protocol:**

1. $[\![\mathsf{is\_permutation}]\!] \leftarrow \mathsf{PermCheck}([\![\mathcal{L}]\!], [\![\mathcal{L}']\!])$
2. For $i \in [S-1]$
   (a) $[\![\alpha_i]\!] \leftarrow [\![\mathcal{L}'[i+1]]\!] - [\![\mathcal{L}'[i]]\!]$
   (b) $[\![\lambda_i]\!] \leftarrow [\![\alpha_i]\!] \cdot [\![1 - \alpha_i]\!]$.
3. Add all the following to the list of outputs:
   – $[\![\mathsf{is\_permutation}]\!]$
   – $[\![\lambda_i]\!]$ for $i \in [S-1]$
   – $[\![\mathcal{L}'[1]]\!] - B_1$
   – $[\![\mathcal{L}'[S]]\!] - B_2$

**Fig. 9.** Bound Check for a batch of sensitive values

### 7.3 Array Access Check

We now describe our version of $C_{\mathsf{check}}$, see Figure 10. We assume the memory has $N$ slots and is first initialized with sensitive values $M_i$. The array $\mathcal{L}$ consists of tuples of the form

$$(\underbrace{\mathsf{memory\_address}}_{\ell}, \underbrace{\mathsf{global\_timestamp}}_{t}, \underbrace{\mathsf{operation}}_{\mathsf{op}}, \underbrace{\mathsf{data}}_{d}).$$

Here, $\ell \in [N]$, $t \in [N+T]$, $\mathsf{op} \in \{0, 1\}$ (0 for read, 1 for write), and $d$ is the data that has been read or written.

*Intuition Behind the Check:* The protocol takes as input the initial array $M$ arranged into a list $\mathcal{L}$ as described before. The list of tuples is sorted first by the address $\ell$, and then by the timestamp $t$, forming a list $\mathcal{L}'$ which consists of contiguous blocks for each address $\ell = 1, \ldots, N$ that list the consecutive accesses to the same address $\ell$ sorting chronologically starting with writing the initial value $M_\ell$.

We need to check the following conditions hold:

– Each block concerns one valid address and all addresses are covered
– Inside each block, the instructions are ordered by their timestamp
– If the operation is $\mathsf{read}$, then the read value matches the previous value at that address

$C_{\mathsf{check}}$

**Input:** $[\![\mathcal{L}]\!] = [(1, 1, 1, [\![M_1]\!]), \dots (N, N, 1, [\![M_N]\!]),$
$([\![l_{N+1}]\!], N+1, [\![\mathsf{op}_{N+1}]\!], [\![d_{N+1}]\!]), \dots, ([\![l_{N+T}]\!], N+T, [\![\mathsf{op}_{N+T}]\!], [\![d_{N+T}]\!])]$
$[\![\mathcal{L}']\!]$ containing entries of $\mathcal{L}$ sorted first by $\ell$ then by $t$.
**Protocol:**

1. $[\![\mathsf{is\_permutation}]\!] \leftarrow \mathsf{PermCheck}([\![\mathcal{L}]\!], [\![\mathcal{L}']\!])$
2. For $i \in [N+T-1]$ do
   (a) $[\![\alpha_i]\!] \leftarrow 1 - ([\![\ell'_{i+1}]\!] - [\![\ell'_i]\!])$
   (b) $[\![\lambda_i]\!] \leftarrow [\![\alpha_i]\!] \cdot [\![1 - \alpha_i]\!]$.
   (c) $[\![\tilde{\tau}_i]\!]_j \leftarrow [\![\alpha_i]\!], [\![t'_{i+1} - t'_i]\!]$ and $[\![\tau_i]\!] \leftarrow [\![\tilde{\tau}_i]\!] + (1 - [\![\alpha_i]\!])$.
   (d) $[\![\zeta_i]\!] \leftarrow [\![\mathsf{op}_i]\!] \cdot [\![1 - \mathsf{op}_i]\!]$.
   (e) $[\![\beta_i]\!] \leftarrow [\![d'_i]\!] - [\![d'_{i+1}]\!]$.
   (f) $[\![\tilde{\gamma}_i]\!] \leftarrow [\![\alpha_i]\!] \cdot [\![\beta_i]\!]$ and $[\![\gamma_i]\!] \leftarrow [\![\tilde{\gamma}_i]\!] \cdot (1 - [\![\mathsf{op}_{i+1}]\!])$.
3. $[\![\mathsf{is\_in\_bound}]\!] \leftarrow \mathsf{BdCheck}((\{[\![\tau_i]\!]\})_{i \in [N+T-1]}, 1, N+T-1)$
4. Add all the following to the list of outputs
   – $[\![\mathsf{is\_permutation}]\!]$
   – $[\![\lambda_i]\!]$ for $i \in [N+T-1]$
   – $[\![\gamma_i]\!]$ for $i \in [N+T-1]$
   – $[\![\zeta_i]\!]$ for $i \in [N+T-1]$
   – $[\![\mathsf{is\_in\_bound}]\!]$
   – $N - [\![\ell'_{N+T}]\!]$

**Fig. 10.** Complete checking circuit for random memory accesses

– Each operation is either a read or a write.

The used variables carry the following meaning:

– $\alpha_i = 1$ if and only if $\ell'_i = \ell'_{i+1}$ and 0 otherwise, i.e., when the next tuple describes an access to the same address
– $\lambda_i = 0$ if and only if $\alpha_i \in \{0, 1\}$
– $\tau_i$ is the difference between the timestamps of subsequent accesses otherwise, $\tau_i = 1$
– $\zeta_i = 0$ if and only if $\mathsf{op}_i \in \{0, 1\}$
– $\beta_i$ is the difference between the data $d'_i - d'_{i+1}$ which is supposed to be 0 if the next tuple is a read instruction
– $\gamma_i = \beta_i$ if and only if $\mathsf{op}_{i+1}$ is a read operation to the same address; therefore it is supposed to be zero.

*Changes Compared to [DOTV22]:* The protocol of [DOTV22] uses the so-called EqCheck circuit, that takes to shared values $[\![x]\!], [\![y]\!]$ and outputs a shared bit $[\![b]\!]$ such that $b = 1$ if and only if $x = y$. We cannot use the EqCheck circuit in our setting, since it relies on the existence of inverses of arbitrary non-zero elements. Hence, we introduce some changes:

– Changes to the $\alpha_i$:
  • Used to be $\mathsf{EqCheck}(\ell'_\mathsf{i}, \ell'_\mathsf{i+1})$.
  • Now is $1 - (\ell'_{i+1} - \ell'_i)$ and check $\alpha_i \in \{0, 1\}$ with $\lambda_i$.
– Changes to the $\beta_i$:
  • Used to be $\mathsf{EqCheck}(\mathsf{d}'_\mathsf{i}, \mathsf{d}'_\mathsf{i+1})$.
  • Now is $d'_i - d'_{i+1}$.

*Zero-knowledge.* Replacing $\alpha_i$ this way does not impact zero-knowledge as for a honest proof, consecutive memory addresses are at most 1 apart. Replacing $\beta_i$ does not impact zero-knowledge either as it only appears in $\gamma_i$ when both $\alpha_i = 1$ and $\mathsf{op}'_{i+1} = 0$ (i.e. read), in which case for an honest proof we expect $\beta_i = 0$.

*Soundness.* Replacing $\alpha_i$ does not impact soundness as it is still an equality check as we ensure $\alpha_i \in \{0,1\}$ with $\lambda_i$. Replacing $\beta_i$ does not impact soundness either as for $\alpha_i = 0$ or $\mathsf{op}'_{i+1} = 1$, we allow $d'_i$ and $d'_{i+1}$ to be arbitrary and when $\alpha_i = 1$ and $\mathsf{op}'_{i+1} = 0$ we deterministically ensure $d'_i = d'_{i+1}$ with the $\Pi_{\mathsf{Zero\text{-}Check}}$ on $\gamma_i$.


### Acknowledgements

The work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085, by CyberSecurity Research Flanders with reference number VR20192203, by the FWO under an Odysseus project GOH9718N, and by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No. 803096 (SPEC). Cyprien Delpech de Saint Guilhem is a Junior FWO Postdoctoral Fellow under project 1266123N. The work of the last author was conducted whilst they were a PhD student at KU Leuven.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the US Government, Cyber Security Research Flanders or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.


## References


ACD+19. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Heidelberg, December 2019.

AHIV17. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

BBC+19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, August 2019.

BBMH+21. Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211. ACM Press, November 2021.

BBMHS22. Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Heidelberg, August 2022.

BDK+21. Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 266–297. Springer, Heidelberg, May 2021.

BN20. Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 495–526. Springer, Heidelberg, May 2020.

CCKP19. Shuo Chen, Jung Hee Cheon, Dongwoo Kim, and Daejun Park. Verifiable computing for approximate computation. Cryptology ePrint Archive, Report 2019/762, 2019. `https://eprint.iacr.org/2019/762`.

CCXY18. Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 395–426. Springer, Heidelberg, August 2018.

CDE+18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod $2^k$ for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

DOT21. Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

DOTV22. Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 615–638, Amalfi, Italy, September 2022. Springer, Heidelberg.

EGK+20. Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

EXY22. Daniel Escudero, Chaoping Xing, and Chen Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 383–412. Springer, Heidelberg, August 2022.

Feh98. Serge Fehr. Span programs over rings and how to share a secret from a module, 1998. MSc Thesis, ETH Zurich.

FMRV22. Thibauld Feneuil, Jules Maire, Matthieu Rivain, and Damien Vergnaud. Zero-knowledge protocols for the subset sum problem from MPC-in-the-head with rejection. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 371–402. Springer, Heidelberg, December 2022.

FR22. Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022. https://eprint.iacr.org/2022/1407.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

GHAH+23. Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *PoPETs*, 2023(1):627–640, January 2023.

GMO16. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

GMR85. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

JSL22. Robin Jadoul, Nigel P. Smart, and Barry Van Leeuwen. MPC for $Q_2$ access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 131–151. Springer, Heidelberg, September / October 2022.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

KZ22. Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Report 2022/588, 2022. https://eprint.iacr.org/2022/588.

LXY23. Fuchun Lin, Chaoping Xing, and Yizhou Yao. More efficient zero-knowledge protocols over $\mathbb{Z}_{2^k}$ via galois rings. Cryptology ePrint Archive, Report 2023/150, 2023. https://eprint.iacr.org/2023/150.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

SS23. Victor Shoup and Nigel P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. Cryptology ePrint Archive, Paper 2023/536, 2023. https://eprint.iacr.org/2023/536.