Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and $\mathbb{Z}_{2^k}$

Carsten Baum
Aarhus University
cbaum@cs.au.dk

Lennart Braun
Aarhus University
braun@cs.au.dk

Alexander Munch-Hansen
Aarhus University
almun@cs.au.dk

Peter Scholl
Aarhus University
peter.scholl@cs.au.dk

June 15, 2021

Abstract

Zero-Knowledge proofs are highly flexible cryptographic protocols that are an important building block for many secure systems. Typically, these are defined with respect to statements that are formulated as arithmetic operations over a fixed finite field. This inflexibility is a disadvantage when it comes to complex programs, as some fields are more amenable to express certain operations than others. At the same time, there do not seem to be many proofs with a programming model similar to those found in modern computer architectures that perform arithmetic with 32 or 64 bit integers.

In this work, we present solutions to both of these problems. First, we show how to efficiently check consistency of secret values between different instances of Zero Knowledge protocols based on the commit-and-prove paradigm. This allows a protocol user to easily switch to the most efficient representation for a given task. To achieve this, we modify the *extended doubly-authenticated bits* (`edaBits`) approach by Escudero *et al.* (Crypto 2020), originally developed for MPC, and optimize it for the Zero-Knowledge setting. As an application of our consistency check, we also introduce protocols for efficiently verifying truncations and comparisons of shared values both modulo a large prime p and modulo 2^k.

Finally, we complement our conversion protocols with new protocols for verifying arithmetic statements in $\mathbb{Z}_{2^k}$. Here, we build upon recent interactive proof systems based on information-theoretic MACs and vector oblivious linear evaluation (VOLE), and show how this paradigm can be adapted to the ring setting. In particular, we show that supporting such modular operations natively in a proof system can be almost as efficient as proofs over large fields or bits, and this also easily plugs into our framework for Zero Knowledge conversions.

1 Introduction

Zero-Knowledge proofs are a cryptographic primitive where a prover convinces a verifier that a statement is true. The verifier should be convinced only of true statements even if the prover is malicious, and moreover, the verifier should not learn anything beyond the fact that the statement holds. Current state-of-the-art Zero-Knowledge (ZK) protocols for arbitrary functions work over either $\mathbb{Z}_p$ for a large p or $\mathbb{Z}_2$. The computation is typically modeled as a circuit of operations that equal the operations of the underlying field, and the efficiency of a proof depends on the number of gates that the circuit has.

A recent line of work has been investigating the scalability of ZK protocols for very large statements, represented as, for instance, circuits with billions of gates. This can be seen in work such as zero-knowledge from garbled circuits [JKO13, FNO15, ZRE15, HK20] and vector oblivious linear evaluation (VOLE) [WYKW20, BMRS20, DIO20, YSWW21]. To handle complex statements, protocols in this setting often have the drawback of requiring more interaction compared with other approaches such as MPC-in-the-head, SNARKs or PCPs, thus sacrificing on proof sizes and public verifiability. However, the advantage is that these protocols typically have lower overhead for the prover, in terms of computational and memory resources, thus scaling better as the statement size increases.

Certain functions are known to be "more efficient" to express as circuits over a specific domain. For example, comparisons or other bit operations are most efficient when expressed over $\mathbb{Z}_2$, while integer arithmetic best fits into $\mathbb{Z}_p$. At the same time, neither of these captures arithmetic modulo 2^k efficiently, which is the standard model of current computer architectures. Most state-of-the-art ZK compilers only operate over a single domain, so for example, if this is $\mathbb{Z}_p$ for a large prime p, then any comparison operation will first require a costly bit-decomposition, followed by emulation of the binary circuit logic in $\mathbb{Z}_p$. If there was instead a way to efficiently *switch* representations, a more suitable protocol over $\mathbb{Z}_2$ could be used instead, for certain parts of the computation.

1.1 Our Contributions

In this work, we address the above shortcomings, by introducing efficient conversion protocols for "commit-and-prove"-type ZK, such as recent VOLE-based protocols. We then build on these conversions by presenting new, high-level gadgets for common operations like truncation and comparison. Finally, we supplement this with efficient ZK protocols for arithmetic circuits over $\mathbb{Z}_{2^k}$, which are also compatible with our previous protocols.

Below, we give a more detailed, technical summary of these contributions.

Commit-and-prove setting. Our protocols work in the commit-and-prove paradigm, where the prover first commits to the secret witness, before proving various properties about it. Assume we have two different commitment schemes, working over $\mathbb{Z}_2$ and $\mathbb{Z}_M$, and denote by $[x]_2$ or $[x]_M$ that the value $x \in \mathbb{Z}_M$ has been committed to in one of the two respective schemes.

Note that our protocols are completely agnostic as to the commitment scheme that is used, provided it is linearly homomorphic. However, in practice a fast instantiation can be obtained using information-theoretic MACs based on recent advances in VOLE [BCGI18, SGRR19, BCG$^+$19a, YWL$^+$20] from the LPN assumption. This has been the approach taken in recent VOLE-based ZK protocols [WYKW20, BMRS20], which exploit the high computational efficiency and low communication overhead of LPN-based VOLE.

Conversions. The goal of our conversion protocol is to verify that a sequence of committed bits $[x_0]_2, \ldots, [x_{m-1}]_2$ correspond to the committed arithmetic value $[x]_M$, where $x = \sum_{i=0}^{m-1} 2^i x_i \bmod M$.

In the MPC setting, Escudero *et al.* [EGK$^+$20] showed how to use *extended doubly-authenticated bits*, or `edaBits`, for this task. `edaBits` are *random* tuples of commitments $(([r_i]_2)_{i=0}^{m-1}, [r]_M)$ that are guaranteed to be consistent. By preprocessing random `edaBits`, [EGK$^+$20] showed how conversions between secret values can then be done efficiently in MPC in an online phase. Note that in MPC, the `edaBits` are actually secret-shared and known to nobody; however, the protocol of [EGK$^+$20] starts by first creating private `edaBits` known to one party, and then summing these up across the parties to obtain secret-shared `edaBits`. In the ZK setting, the prover knows the values of the `edaBits`, so the second phase can clearly be omitted.

With this observation, a straightforward application of `edaBits` leads to the following basic conversion protocol between prover $\mathcal{P}$ and verifier $\mathcal{V}$:

1. $\mathcal{P}$ commits to $[x_0]_2, \ldots, [x_{m-1}]_2$.
2. $\mathcal{P}$ and $\mathcal{V}$ run the `edaBits` protocol to generate a valid committed `edaBit` $([r_0]_2, \ldots, [r_{m-1}], [r]_M)$.
3. $\mathcal{P}$ uses $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)$ to convert $[x_0]_2, \ldots, [x_{m-1}]_2$ into $[x]_M$ correctly.

In the last step, $\mathcal{P}$ will first commit to $[x]_M$, then open $[x+r]_M$ to $\mathcal{V}$, and finally prove that $x+r$ equals the sum of the committed bits $[x_i]_2$ and $[r_i]_2$. The latter check requires the verification of a binary circuit for addition modulo M over $\mathbb{Z}_2$.

In our protocol, we introduce several optimizations of this approach, tailored to the ZK setting. Firstly, we observe that in the ZK setting, it is not necessary to create *random* verified `edaBits`, if we can instead just apply the `edaBit` verification protocol to the actual conversion tuples $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$, from the witness. This change would remove the need for the binary addition circuits in the last step. Unfortunately, the protocol of [EGK+20] cannot be used for this setting, as it uses a cut-and-choose procedure where a small fraction of `edaBits` are opened and then discarded, which may leak information on our conversion tuples. Instead, we present a new `edaBit` consistency check where the cut-and-choose step does not leak on the secret conversion tuples used as input, essentially by replacing the uniformly random permutation of `edaBits` with a permutation sampled from a more restricted set. This requires a careful analysis to show that the modified check still has a low enough cheating probability.

In addition, we present a simplification of the protocol which further reduces communication, by using "faulty `daBits`". `daBits` (doubly-authenticated bits) are length-1 `edaBits` $([x]_2, [x]_M)$, which are used in the consistency check of [EGK+20] when M is not a power of 2. However, producing a correct `daBit` requires proving that $[x]_M$ is a bit, introducing an extra check. We show that our protocol with a slight modification remains secure even when the `daBits` may be inconsistent. Essentially, this boils down to showing that any errors in faulty `daBits` can be translated into equivalent errors in the binary addition circuit used to check the `edaBits`. Since our basic protocol is already resilient to faulty addition circuits, the same security analysis applies.

Comparison & Truncation. Using our efficient conversion check, we give new protocols for verifying integer truncation and integer comparison on committed values. A natural starting point would be to adapt the MPC protocols in [EGK+20], which also used `edaBits` for these operations. However, a drawback of these protocols is that in addition to `edaBits`, they use auxiliary binary comparison circuits, which add further costs. We show that in the ZK setting, these can be avoided, and obtain protocols which only rely on our efficient conversion check.

As a building block of our protocols, we make use of the fact that our `edaBit` consistency check can easily be used to prove that a committed value $x \in \mathbb{Z}_M$ is at most m bits in length, for some public m. We then show that integer truncation in $\mathbb{Z}_M$ can be decomposed into just two length checks, by exploiting the fact that the prover can commit to arbitrary values dependent on the witness. Then, given truncation, we can easily obtain a comparison check, which shows that a committed bit $[b]_2$ encodes $b = (x \overset{?}{<} y)$, where $[x]_M, [y]_M$ are committed.

ZK for Arithmetic Circuits over $\mathbb{Z}_{2^k}$. Our conversion, truncation and comparison protocols can all be made to work with either a field $\mathbb{Z}_p$, or a ring $\mathbb{Z}_{2^k}$, giving flexibility in high-level applications. While ZK protocols for $\mathbb{Z}_p$ and $\mathbb{Z}_2$ have been well-studied, there is less work on protocols for circuits over $\mathbb{Z}_{2^k}$, especially in the commit-and-prove setting. We take the first steps towards this, by showing how to use VOLE-based information-theoretic MACs for ZK over $\mathbb{Z}_{2^k}$, by adapting the techniques from SPD$\mathbb{Z}_{2^k}$ [CDE+18]. Given the MACs, which serve as homomorphic commitments in $\mathbb{Z}_{2^k}$, we show how to efficiently verify multiplications on committed values. We present two possible approaches: the first is based on a simple cut-and-choose procedure, adapted from [WYKW20] for binary circuits; in the second approach, we adapt the field-based multiplication check from [BMRS20] to work over rings, which requires some non-trivial modifications.

Since these protocols use VOLE-based information-theoretic MACs, we obtain ZK protocols in the preprocessing model, assuming a trusted setup to distribute VOLE (or short seeds which expand to VOLE [BCGI18]). Removing the trusted setup can be done with an actively secure VOLE protocol over $\mathbb{Z}_{2^k}$. We note that the LPN-based construction of [BCGI18] also works over $\mathbb{Z}_{2^k}$ (as implemented in [SGRR19]), although currently only with passive security. It is an interesting future direction to extend efficient actively secure protocols [BCG+19a, WYKW20] to the $\mathbb{Z}_{2^k}$ setting.

Concrete Efficiency. We analyze the efficiency of our protocols, in terms of the bandwidth requirements and the amount of VOLE or OT preprocessing that is needed. Compared with a baseline protocol consisting of a straightforward application of `edaBits` [EGK+20] to ZK, our optimized conversion protocol reduces communication by more than 2x, while also reducing the number of VOLEs required by around 4x. Since our protocol does not introduce any extra computation, we expect the concrete performance gains in an implementation to be similar (depending on the network setting).

Our $\mathbb{Z}_{2^k}$ protocols achieve amortized communication costs of a single ring element to open a commitment, and two elements of the larger ring $\mathbb{Z}_{2^{k+s}}$, where s is a statistical security parameter, plus two VOLEs, to verify a multiplication. The former is optimal, and the latter is competitive with state-of-the-art protocols for large fields such as [WYKW20, BMRS20], which need to transfer 2–3 field elements.

1.2 Related Work

Recently and independently, Quicksilver [YSWW21] proposed a new VOLE-based ZK protocol for boolean and arithmetic circuits (without support for conversions). Unlike the protocols in this work, Quicksilver makes non-black-box use of information-theoretic MACs to improve efficiency, obtaining a cost of just 1 field element per multiplication gate when evaluating a circuit. Since the core of our protocol uses potentially faulty information to verify `edaBits`, it seems likely that the techniques from Quicksilver could be plugged in to avoid having to deal with these faulty components, which would simplify much of our security analysis. On the down side, since Quicksilver makes non-black-box use of VOLE-based MACs, it would not be applicable in settings based on other types of homomorphic commitments, or applications such as proofs of disjunctions in [BMRS20], which assumes a black-box commitment scheme. Thus, our protocol is more general and may be of use in a wider range of applications. We leave a more detailed exploration of combining Quicksilver and our techniques as future work.

Another related work is Rabbit [MRVW21], which provided improved protocols for secure comparison and truncation based on `edaBits`, in the MPC setting. Similarly to our work in ZK, Rabbit allows to avoid the large "gap" between the field size and the desired message space when running these protocols; however, our techniques in the ZK setting are different.

In LegoSNARK [CFQ19] the authors show how to combine different succinct ZK proof systems. Our work differs as we focus on the setting where data is represented in different rings of possibly constant size for each subtask, whereas [CFQ19] relies on large groups.

2 Preliminaries

In this section we introduce several primitives which are used throughout the constructions in this paper.

2.1 Notation

We use M to denote a modulus which is either a large prime p, or 2^k. As a short hand, $\equiv_k$ denotes equality modulo 2^k. We use $[x]_M$ or $[x]_2$ to denote authenticated values (see Sections 2.3 and 2.4) from the plaintext space $\mathbb{Z}_M$ or $\mathbb{Z}_2$, and write just $[x]$ when the modulus is clear from the context. We let s denote a statistical security parameter and $[n]$ denote the set $\{1, \ldots, n\}$.

2.2 Zero-Knowledge Proofs

Zero-knowledge proofs (of knowledge) are interactive two-party protocols that allow the prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a certain statement is true (and that it possesses a witness to this fact). This happens in a way such that $\mathcal{V}$ does not learn anything else besides this fact that it could not compute by itself. Instead of using the classical definition by Goldwasser et al. [GMR85], we define zero-knowledge using an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ for satisfiability of circuits $\mathcal{C}$: On input $(\mathsf{Prove}, \mathcal{C}, w)$ from $\mathcal{P}$ and $(\mathsf{Prove}, \mathcal{C})$ from $\mathcal{V}$ the functionality $\mathcal{F}_{\mathsf{ZK}}$ outputs $\top$ to $\mathcal{V}$ iff. $\mathcal{C}(w) = 1$ holds, and sends $\perp$ otherwise [WYKW20].

Following previous works (e.g. [WYKW20, BMRS20]), we use the commit-and-prove strategy to instantiate $\mathcal{F}_{\mathsf{ZK}}$ using homomorphic commitments (see Section 2.4). These allow the prover $\mathcal{P}$ to commit to its witness w. Then the circuit $\mathcal{C}$ can be evaluated on the committed witness to obtain a commitment to the output, which is opened to prove that indeed $\mathcal{C}(w) = 1$ holds.

2.3 VOLE and Linearly Homomorphic MACs

Oblivious transfer (OT) [EGL82] is a two-party protocol, where the receiver can obliviously inputs a bit b to choose between two messages m_0, m_1 held by the sender to obtain m_b. In *correlated* OT (COT) [ALSZ13] the messages are chosen randomly given a sender-specified correlation function, e.g. $x \mapsto x + \delta$ such that $m_1 = m_0 + \delta$ holds over some domain. Thus, the receiver obtains $m_b = \delta \cdot b + m_0$.

While OT inherently requires relatively costly public key cryptography [IR89], OT extension [IKNP03] allows to expand a small number of regularly computed OTs into a large number of OTs using only relatively cheap symmetric key cryptography.

Oblivious linear-function evaluation (OLE) [NP99, IPS09] is an arithmetic generalization of COT allowing a receiver to evaluate a secret linear equation $\alpha \cdot X + \beta$ (over a field $\mathbb{F}_p$ or ring $\mathbb{Z}_{2^k}$) held by the sender at a point of its choice x to obtain $y = \alpha \cdot x + \beta$. This can be extended into *vector* OLE (VOLE) [ADI+17] where x and β are vectors of the same length rather than single field elements. *Subfield* VOLEs [BCG+19b] furthermore extends this concept such that the elements of α and β live in an extension field $\mathbb{F}_{p^r} \supset \mathbb{F}_p$. Random (subfield) VOLE, where the inputs are chosen randomly by the functionality, is easier to realize and can be used to instantiate normal VOLE, by sending correction values.

We use *information-theoretic message authentication codes* (MACs) to authenticate values in finite fields $\mathbb{Z}_p$ and rings $\mathbb{Z}_{2^k}$. The case $\mathbb{Z}_{2^k}$ is discussed in Section 5.2 where we adapt the work of [CDE+18] to the zero-knowledge setting. For fields $\mathbb{Z}_p$, we use BeDOZa-style MACs [BDOZ11] which can be generated as follows: To authenticate values $x_1, \ldots, x_n \in \mathbb{Z}_p$ known to $\mathcal{P}$, random keys $\Delta, K[x_1], \ldots, K[x_n] \in_R \mathbb{Z}_p$ are chosen by $\mathcal{V}$, and then $\mathcal{P}$ obtains the MACs $M[x_i] \leftarrow \Delta \cdot x_i + K[x_i] \in \mathbb{Z}_p$. We use the notation $[x_i]_p$ for this. To open $[x]_p$, $\mathcal{P}$ sends x and $M[x]$ to $\mathcal{V}$, who checks that $M[x] = \Delta \cdot x + K[x]$ holds. These authentications are linearly homomorphic: Given authenticated values $[x]_p$ and $[y]_p$ and public values a, b, $\mathcal{P}$ and $\mathcal{V}$ can locally compute $[z]_p$ for $z := a \cdot x + y + b$ by setting $M[z] := a \cdot M[x] + M[y]$ and $K[z] := a \cdot K[x] + K[y] - \Delta \cdot b$. For large enough p, this is secure since forgery would imply correctly guessing a random element of $\mathbb{Z}_p$. For smaller p, the keys Δ and $K[x_i]$ are instead chosen from an extension field $\mathbb{Z}_{p^r}$ such that p^r is large enough. The MACs can be efficiently computed with (subfield) VOLE [WYKW20, BMRS20].

2.4 Homomorphic Commitment Functionality

As discussed in Section 2.2, we use the commit-and-prove paradigm for our zero-knowledge protocols. To this end, we define a commitment functionality. It allows the prover $\mathcal{P}$ to commit to values, and choose to reveal them at a later point in time, such that the verifier $\mathcal{V}$ is convinced that the values had not been modified in the meantime. Moreover, the functionality allows to perform certain operations of the underlying algebraic structure on the committed values, and to check if these satisfy certain relations.

The commitment functionality can be instantiated using linearly homomorphic information-theoretic MACs (see Section 2.3). For finite fields $\mathbb{Z}_p$, this was shown with the protocols Wolverine [WYKW20] and Mac'n'Cheese [BMRS20]. We refer to their works for details. For rings $\mathbb{Z}_{2^k}$, we present an instantiation in Section 5.2.

We formally define the homomorphic commitments using the ideal functionality $\mathcal{F}^R_{\mathsf{ComZK}}$ given in Figure 1. The parameter R denotes the message space, which is in our case either a ring $\mathbb{Z}_{2^k}$ or a field $\mathbb{Z}_p$. In addition to the common Input and Open operations, which enables $\mathcal{P}$ to commit to a value and reveal it to $\mathcal{V}$ at a later point, we also model Random and CheckZero, for generating commitments of random values and verifying that a committed value equals zero, respectively, which enables more efficient implementations. Moreover, $\mathcal{F}^R_{\mathsf{ComZK}}$ allows via Affine to compute affine combinations of committed values with public coefficients yielding again a commitment of the result. Finally, CheckMult allows to verify that a set of triples satisfy a multiplicative relation, i.e. for each triple, the third commitment contains the product of the first two committed values.

Since the commitment functionality is based on information-theoretic MACs, we use the same notation $[x]$ to denote a committed value $x \in R$. We use this shorthand to to simplify the presentation of higher-level protocols without explicitly mentioning the commitment identifiers. We use also shorthands for the different methods of $\mathcal{F}^R_{\mathsf{ComZK}}$, e.g. we write something like $[z] \leftarrow a \cdot [x] + [y] + b$ when invoking the Affine method (see Figure 1). We write $[x]_M$, if the domain $\mathbb{Z}_M$ of the committed values is not clear from the context, or if we have to distinguish commitments over multiple different domains.

Homomorphic Commitment Functionality $\mathcal{F}^{R}_{\mathsf{ComZK}}$

The functionality communicates with two parties $\mathcal{P}, \mathcal{V}$ as well as an adversary $\mathcal{S}$ that may corrupt either party. $\mathcal{S}$ may at any point send a message (abort), upon which $\mathcal{F}^{R}_{\mathsf{ComZK}}$ sends (abort) to all parties and terminates. $\mathcal{F}^{R}_{\mathsf{ComZK}}$ contains a state st that is initially $\emptyset$.

Random On input $(\mathsf{Random}, \mathsf{id})$ from $\mathcal{P}, \mathcal{V}$ and where $(\mathsf{id}, \cdot) \notin \mathsf{st}$:

1. If $\mathcal{P}$ is corrupted, obtain $x_{\mathsf{id}} \in R$ from $\mathcal{S}$. Otherwise sample $x_{\mathsf{id}} \in_R R$ uniformly at random.
2. Set $\mathsf{st} \leftarrow \mathsf{st} \cup \{(\mathsf{id}, x_{\mathsf{id}})\}$ and send x_{id} to $\mathcal{P}$.

We use the shorthand $[x] \leftarrow \mathsf{Random}()$.

Affine Combination On input $(\mathsf{Affine}, \mathsf{id}_o, \mathsf{id}_1, \dots, \mathsf{id}_n, \alpha_0, \dots, \alpha_n)$ from $\mathcal{P}, \mathcal{V}$ where $(\mathsf{id}_i, x_{\mathsf{id}_i}) \in \mathsf{st}$ for $i = 1, \dots, n$ and $(\mathsf{id}_o, \cdot) \notin \mathsf{st}$:

1. Set $x_{\mathsf{id}_o} \leftarrow \alpha_0 + \sum_{i=1}^{n} \alpha_1 \cdot x_{\mathsf{id}_i}$ and $\mathsf{st} \leftarrow \mathsf{st} \cup \{(\mathsf{id}_o, x_{\mathsf{id}_o})\}$.

We use shorthands such as $[z] \leftarrow a \cdot [x] + [y] + b$.

CheckZero On input $(\mathsf{CheckZero}, \mathsf{id}_1, \dots, \mathsf{id}_n)$ from $\mathcal{P}, \mathcal{V}$ and where $(\mathsf{id}_i, x_{\mathsf{id}_i}) \in \mathsf{st}$ for $i = 1, \dots, n$:

1. If $x_{\mathsf{id}_1} = \cdots = x_{\mathsf{id}_n} = 0$, then send $(\mathsf{success})$ to $\mathcal{V}$, otherwise send (abort) to all parties and terminate.

We use the shorthand $\mathsf{CheckZero}([x_1], \dots, [x_n])$.

Input On inputs $(\mathsf{Input}, \mathsf{id}, x_{\mathsf{id}})$ from $\mathcal{P}$ and $(\mathsf{Input}, \mathsf{id})$ from $\mathcal{V}$ and where $(\mathsf{id}, \cdot) \notin \mathsf{st}$:

1. Set $\mathsf{st} \leftarrow \mathsf{st} \cup \{(\mathsf{id}, x)\}$.

We use the shorthand $[x] \leftarrow \mathsf{Input}(x)$.

Open On input $(\mathsf{Open}, \mathsf{id}_1, \dots, \mathsf{id}_n)$ from $\mathcal{P}, \mathcal{V}$ where $(\mathsf{id}_i, x_{\mathsf{id}_i}) \in \mathsf{st}$ for $i = 1, \dots, n$:

1. Send $x_{\mathsf{id}_1}, \dots, x_{\mathsf{id}_n}$, to $\mathcal{V}$.

We use the shorthand $x_1, \dots, x_x \leftarrow \mathsf{Open}([x_1], \dots, [x_n])$. Moreover, we might use the following macro: $x \leftarrow \mathsf{Open}([x], \mathsf{lst})$ denotes that $\mathcal{P}$ sends x to $\mathcal{V}$ and they add $[x] - x$ to the list lst.

MultiplicationCheck Upon $\mathcal{P}$ & $\mathcal{V}$ inputting $(\mathsf{CheckMult}, (\mathsf{id}_{x,i}, \mathsf{id}_{y,i}, \mathsf{id}_{z,i})_{i=1}^{n})$ where $(\mathsf{id}_{x_i}, x_i), (\mathsf{id}_{y_i}, y_i), (\mathsf{id}_{z_i}, z_i) \in \mathsf{st}$ for $i = 1, \dots, n$:

1. Send $(\mathsf{success})$ to $\mathcal{V}$ if $x_i \cdot y_i = z_i$ holds for all $i = 1, \dots, n$, otherwise send (abort) to all parties and terminate.

We use the shorthand $\mathsf{CheckMult}(([x_i], [y_i], [z_i])_{i=1}^{n})$.

Figure 1: Ideal functionality modeling homomorphic commitments of values in the ring (or field) R of the prover $\mathcal{P}$ to the verifier $\mathcal{V}$.

2.5 Extended Doubly-Authenticated Bits

A *doubly-authenticated bit* (or `daBit` for short) is a bit b that is authenticated in both a binary and arithmetic domain, i.e. a tuple $([b]_2, [b]_M)$. `daBits` can be used to convert a single bit from the binary to the arithmetic domain or vice versa [RW19, MRVW21].

Their generalization, called `edaBits` (due to Escudero *et al.* [EGK+20]), is defined as m bits $b_0, \ldots, b_{m-1}$ which are each authenticated in the binary domain while their sum is authenticated in the arithmetic one, i.e. $([b_0]_2, \ldots, [b_{m-1}]_2, [b]_M)$, for some $m \leq \lceil \log M \rceil$. These `edaBits` allow for optimised conversions of authenticated values, and allow to securely compute truncations or extract the most significant bit of a secret value in MPC.

We now quickly recap their `edaBits` generation protocol (originally defined in the multi-party computation context) as we build upon their construction later. The construction of [EGK+20] consists of two different phases: in the first phase, each party locally samples `edaBits` and proves to all other parties that they were computed correctly. Then, in a second phase, these local contributions are combined to global, secret `edaBits`. In our setting however, only the prover will use `edaBits`, thus it is clear that the second phase can be omitted. Our sampling protocol will only have to ensure that each `edaBit` $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ is indeed consistent, i.e. that $x = \sum_{i=0}^{m-1} x_i 2^i \bmod M$.

The first phase of the `edaBit` sampling routine of [EGK+20] then works as follows (when adapted to the Zero-Knowledge setting):

1. The prover locally samples $(NB + C)m$ bits $r_{i,j}$ for $j \in [NB + C]$ and $i \in \{0, \ldots, m-1\}$. It then combines these into the $NB + C$ values $r_j \leftarrow \sum_{i=0}^{m-1} 2^i r_{i,j}$ yielding `edaBits` $\{(r_{i,j})_{i=0}^{m-1}, r_j\}_{j \in [NB+C]}$.

2. The prover then commits to the binary values $r_{i,j}$ over $\mathbb{Z}_2$ and to the combined values r_j over $\mathbb{Z}_M$.

3. The prover and verifier engage in a check that ensures that the committed values of the prover are consistent. For this, the prover first opens C of the $NB + C$ committed tuples to show consistency (where the choice is made by the verifier). Then the NB `edaBits` are distributed into N buckets of size B. $B - 1$ of the `edaBits` are then used to verify that the remaining `edaBit` per bucket is consistent without leaking information about it.

4. If the check passes, then the remaining `edaBit` in each of the N buckets is known to be consistent.

The main challenge in this protocol is the bucket check in the penultimate step; [EGK+20] show that certain consistency checks can be performed in an unreliable manner, while still being hard to cheat overall, which leads to a complicated analysis.

3 Conversions between $\mathbb{Z}_2$ and $\mathbb{Z}_M$

In this section we present our protocol for performing proofs of consistent conversions in mixed arithmetic-binary circuits that will work with *any* such ZK protocol as described in the preliminaries.

3.1 Conversions and `edaBits` in ZK

In secure multi-party computation, `edaBits` are used to compute a conversion of a value $[x]_M$ that is secret-shared among multiple parties. In the zero-knowledge setting, the prover knows the underlying value x, so there is no need to convert $[x]_M$ securely into its bit decomposition $([x_0]_2, \ldots, [x_{m-1}]_2)$ online. Instead, the prover can commit to $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ in advance, which would itself form a valid `edaBit` *if the conversion is correct.* We call the inputs and outputs of conversion operations *conversion tuples.*

Definition 1 (Conversion Tuple). *Let $M \in \mathbb{N}^+$, $m \leq \lceil \log_2(M) \rceil$, $x \in \mathbb{Z}_M$ and $x_i \in \mathbb{Z}_2$. Then the tuple $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ is called a* conversion tuple. *We call $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ consistent iff $x = \sum_{i=0}^{m-1} 2^i x_i \bmod M$.*

Our conversion protocol in this section provides an efficient way to verify that a large batch of conversion tuples are consistent, i.e. that the committed values are indeed valid `edaBits`. We note that an alternative approach would be to directly apply the method of [EGK+20] — here, first a set of *random*, verified conversion tuples is created, and then one of these is used to check the actual conversion tuple in an online phase. Unfortunately, this online phase check itself involves verifying a binary circuit for addition mod M, which introduces additional expense. We therefore design a new protocol to avoid this, with further optimizations.

Our protocols perform conversions on committed values in $\mathbb{Z}_2$ and $\mathbb{Z}_M$, where we recall that M is either a large prime or 2^k. We model these commitments using the functionality $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ in Figure 2, which extends two instances of $\mathcal{F}_{\mathsf{ComZK}}^{R}$ for $R = \mathbb{Z}_2$ and $R = \mathbb{Z}_M$ and simply parses all method calls to the respective instance.

Functionality $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$

$\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ communicates with two parties $\mathcal{P}, \mathcal{V}$. It contains two separate instances of the commitment functionality $\mathcal{F}_{\mathsf{ComZK}}^{R}$, one for $R = \mathbb{Z}_2$ and the other for $R = \mathbb{Z}_M$. Commitments are denoted as $[\cdot]_2$ and $[\cdot]_M$, respectively.
The parties can use the functions of $\mathcal{F}_{\mathsf{ComZK}}^{R}$ with respect to both domains $\mathbb{Z}_2$ and $\mathbb{Z}_M$, so all functions are parameterized by a domain unless apparent from context. Then, any use of $[\cdot]_2$ or $[\cdot]_M$ interfaces are dealt with in the same way as $\mathcal{F}_{\mathsf{ComZK}}^{R}$.

Figure 2: Ideal functionality modeling communication using commitments over multiple domains.

Finally, we define the ideal functionality for verifying conversions $\mathcal{F}_{\mathsf{Conv}}$ in Figure 3. This functionality extends $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ with a single method VerifyConv. It essentially checks whether or not the two representations of some hidden value are consistent.

Functionality $\mathcal{F}_{\mathsf{Conv}}$

$\mathcal{F}_{\mathsf{Conv}}$ extends the existing functionality $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$, thus containing two commitment instances:

1. $[\cdot]_2$ allows to commit to values from $\mathbb{Z}_2$; and
2. $[\cdot]_M$ allows to commit to values from $\mathbb{Z}_M$,

plus the interface **VerifyConv**. It is assumed that the id's used for VerifyConv have been used with the respective instance of Input prior to calling this method.

VerifyConv: Upon $\mathcal{P}$ and $\mathcal{V}$ inputting $(\mathsf{VerifyConv}, N, m, \{(\mathsf{id}_0^{(j)}, \dots, \mathsf{id}_{m-1}^{(j)}), \mathsf{id}^{(j)}\}_{j \in [N]})$:

1. If $c^{(j)} = \sum_{i=0}^{m-1} 2^i c_i^{(j)}$ for all $j \in [N]$ then output $(\mathsf{success})$ to $\mathcal{V}$, otherwise output abort.

Figure 3: Functionality $\mathcal{F}_{\mathsf{Conv}}$ checking `edaBits`

3.2 The Conversion Verification Protocol Π_{Conv}

The following protocol Π_{Conv} verifies the correctness of a batch of N conversion tuples. Π_{Conv} uses $\mathcal{F}_{\mathsf{Dabit}}$ (Figure 4) to verify correctness of `daBits` (recall, a `daBit` is an `edaBit` of length 1), which is needed in one stage of the protocol. Later, we show how to remove most of the `daBit` check to improve efficiency.

Π_{Conv} also uses multiplication triples, namely, random values $[x]_2, [y]_2, [z]_2$ where $z = x \cdot y$; one multiplication triple can then be used to verify a multiplication on committed inputs at a cost of two openings in $\mathbb{Z}_2$, using a standard technique. In our case, however, we allow the prover to choose all the triples, without verifying their consistency.

Functionality $\mathcal{F}_{\mathsf{Dabit}}$

This functionality extends $\mathcal{F}^{2,M}_{\mathsf{ComZK}}$ with the extra function VerifyDabit that takes a set of IDs $\{(\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]}$ and verifies that $b^{\mathsf{id}^{0,j}} = b^{\mathsf{id}^{1,j}}$ where $b^{\mathsf{id}^{0,j}} \in \mathbb{Z}_2$ an $b^{\mathsf{id}^{1,j}} \in \mathbb{Z}_M$ for all $j \in [N]$. It is assumed that the id's have been Input prior to calling this method.

Verify: On input $(\mathsf{VerifyDabit}, N, \{(\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]})$ by $\mathcal{P}$ and $\mathcal{V}$ where $(\mathsf{id}^{0,j}, b^{\mathsf{id}^{0,j}}), (\mathsf{id}^{1,j}, b^{\mathsf{id}^{1,j}}) \in \mathsf{st}$.

1. If $b^{\mathsf{id}_{0,j}} = b^{\mathsf{id}_{1,j}}$ for all $j \in [N]$, then output (success) to $\mathcal{V}$, otherwise output abort.

Figure 4: Functionality $\mathcal{F}_{\mathsf{Dabit}}$ checking `daBits`.

On a high level, Π_{Conv}, in Figure 5, consists of three phases:

1. Initially, $\mathcal{P}$ commits to auxiliary random `edaBits`, `daBits` and multiplication triples necessary for the check. The `daBits` are verified separately, and then $\mathcal{V}$ chooses a random permutation.

2. After permuting the `edaBits` and multiplication triples, both parties run an implicit cut-and-choose phase. Here, $\mathcal{P}$ opens C of the `edaBits` and triples, which are checked by $\mathcal{V}$.

3. We place each conversion tuple into one of N buckets, each of which contains a conversion tuple $([c_0]_2, \ldots, [c_{m-1}]_2, [c]_M)$, and a set of B `edaBits` $\{([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)_i\}_{i=0}^{B-1}$. None of these have been proven consistent, but C `edaBits` coming from the same pool have been opened in the previous step. Now, over B iterations the prover and verifier for each $j \in [B]$ compute $[c + r_j]_M = [c]_M + [r_j]_M$ and use an addition circuit to check that $([e_0]_2, \ldots, [e_m]_2) = ([c_0]_2, \ldots, [c_{m-1}]_2) + ([r_0]_2, \ldots, [r_{m-1}]_2)$. The addition circuit is evaluated using the multiplication triples (which also may be inconsistent).

For the checks within each bucket, we use the two sub-protocols convertBit2A (Figure 6) and bitADDcarry (Figure 7). The former converts an authentication of a bit $[b]_2$ into an arithmetic authentication $[b]_M$ while the latter adds two authenticated values $([x_0]_2, \ldots, [x_{m-1}]_2)$ and $([y_0]_2, \ldots, [y_{m-1}]_2)$. This uses a ripple-carry adder circuit, which satisfies the following weak tamper-resilient property, as observed in [EGK+20].

Definition 2. *A binary circuit $C : \mathbb{Z}_2^{2m} \to \mathbb{Z}_2^{m+1}$ is weakly additively tamper resilient, if given any additively tampered circuit C^*, obtained by flipping the output of any fixed number of AND gates in C, one of the following two properties hold:*

1. $\forall (x, y) \in \mathbb{Z}_2^{2m} : C(x, y) = C^*(x, y)$; *or*
2. $\forall (x, y) \in \mathbb{Z}_2^{2m} : C(x, y) \neq C^*(x, y)$

Note that the type of additive tampering in Definition 2 models the errors induced by faulty multiplication triples, when used to evaluated a circuit in ZK or MPC. Intuitively, the definition says that the output of the tampered circuit is either incorrect on every possible input or equivalent to the original un-tampered circuit. This gives us the property that an adversary cannot pass the verification protocol using a tampered circuit with both a good conversion tuple and a bad one. Thus, if any provided multiplication triples are incorrect, then the check at those positions would only pass with either a good or a bad conversion tuple (or `edaBit`), but not both.

Protocol Π_{Conv}

Assume that $\mathcal{F}_{\mathsf{Dabit}}$ contains N committed conversion tuples $\{[c_0^{(i)}]_2, \ldots, [c_{m-1}^{(i)}]_2, [c^{(i)}]_M\}_{i \in [N]}$.

// $\mathcal{P}$ commits auxiliary values for conversion check. `daBits` are then verified.

1. $\mathcal{P}$ commits to the following values using $\mathcal{F}_{\mathsf{Dabit}}$:
   - (a) Random `edaBits` $([r_0^{(j)}]_2, \ldots, [r_{m-1}^{(j)}]_2, [r^{(j)}]_M)_{j \in [NB+C]}$.
   - (b) Random `daBits` $([b^{(j)}]_2, [b^{(j)}]_M)_{j \in [NB]}$.
   - (c) Random multiplication triples $([x^{(j)}]_2, [y^{(j)}]_2, [z^{(j)}]_2)_{j \in [NBm+Cm]}$
2. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{VerifyDabit}, NB, \{([b^{(j)}]_2, [b^{(j)}]_M)\}_{j \in [NB]})$ to $\mathcal{F}_{\mathsf{Dabit}}$.

// $\mathcal{P}$ and $\mathcal{V}$ shuffle the auxiliary values and a subset gets opened and verified.

3. $\mathcal{V}$ samples uniformly random permutations $\pi_1 \in S_{NB+C}, \pi_2 \in S_{NB}, \pi_3 \in S_{NBm+Cm}$ and sends them to $\mathcal{P}$.
4. Both parties shuffle the `edaBits` $[r_0^{(j)}]_2, \ldots, [r_{m-1}^{(j)}]_2, [r^{(j)}]_M$ locally according to π_1. They then shuffle $[b_2^{(j)}]_2, [b_M^{(j)}]_M$ according to π_2 and $[x^{(j)}]_2, [y^{(j)}]_2, [z^{(j)}]_2$ according to π_3.
5. Run a cut-and-choose procedure as follows:
   - (a) $\mathcal{P}$ opens $\{[r_0^{(j)}]_2, \ldots, [r_{m-1}^{(j)}]_2, [r^{(j)}]_M\}_{j=NB+1}^{NB+C}$ (the last C `edaBits`) towards $\mathcal{V}$, who in turn checks that $r^{(j)} \overset{?}{=} \sum_{i=0}^{m-1} 2^i \cdot r_i^{(j)}$.
   - (b) $\mathcal{P}$ opens the x, y values for the last Cm triples $\{[x^{(j)}]_2, [y^{(j)}]_2\}_{j=NBm+1}^{NBm+Cm}$ and proves to $\mathcal{V}$ that $\mathsf{CheckZero}([z^{(j)}] - x^{(j)} \cdot y^{(j)})$ for all opened triples.

// $\mathcal{P}$ and $\mathcal{V}$ verify each conversion tuple in a bucket.

6. For the i'th conversion tuple $[c_0]_2, \ldots, [c_{m-1}]_2, [c]_M$, do the following for $j \in [B]$:
   - (a) Let $[r_0]_2, \ldots, [r_{m-1}]_2, [r]_M$ be the $(i-1) \cdot B + j$'th `edaBit` and $[c+r]_M = [c]_M + [r]_M$.
   - (b) Let $([e_0]_2, \ldots, [e_m]_2) \leftarrow$ $\mathsf{bitADDcarry}([c_0]_2, \ldots, [c_{m-1}]_2, [r_0]_2, \ldots, [r_{m-1}]_2)$.
   - (c) Convert $[e_m]_M \leftarrow \mathsf{convertBit2A}([e_m]_2)$ using the $(i-1) \cdot B + j$'th `daBit` $([b]_2, [b]_M)$.
   - (d) Let $[e']_M \leftarrow [c+r]_M - 2^m \cdot [e_m]_M$.
   - (e) Let $e_i \leftarrow \mathsf{Open}([e_i]_2)$ for $i = 0, \ldots, m-1$. Then run $\mathsf{CheckZero}([e']_M - \sum_{i=0}^{m-1} 2^i \cdot e_i)$.
7. If any of the checks fail, $\mathcal{V}$ outputs abort. Otherwise it outputs $(\mathsf{success})$.

Figure 5: Protocol Π_{Conv} to verify Conversion Tuples

Procedure convertBit2A

Input A `daBit` $([r]_2, [r]_M)$ and a commitment $[x]_2$.

Protocol

1. $c \leftarrow \mathsf{Open}([r]_2 \oplus [x]_2)$.
2. Output $[x]_M \leftarrow c + [r]_M - 2 \cdot c \cdot [r]_M$.

Figure 6: Procedure to convert bit from $\mathbb{Z}_2$ to $\mathbb{Z}_M$.

Procedure bitADDcarry

Input Commitments $[x_0]_2, \ldots, [x_{m-1}]_2, [y_0]_2, \ldots, [y_{m-1}]_2$.

Protocol Let $c_0 = 0$.

1. Compute $[c_{i+1}]_2 = [c_i]_2 \oplus (([x_i \oplus c_i]_2) \wedge ([y_i \oplus c_i]_2)), \forall i \in \{0, \ldots, m-1\}$
2. Output $[z_i]_2 = [x_i \oplus y_i \oplus c_i]_2, \forall i \in \{0, \ldots, m-1\}$ and $[c_m]_2$.

Figure 7: A ripple-carry adder

While bitADDcarry will ensure that (assuming correct triples) $([e_0]_2, \ldots, [e_m]_2)$ are computed as required, care must be taken regarding $[c + r_j]_M$ as this may not be representable by m bits any longer (but rather $m+1$). To remedy this, we use a `daBit` to convert $[e_m]_2$ into an arithmetic authentication $[e_m]_M$ to remove the carry from $[c+r_j]_M$ by computing $[e']_M = [c+r_j]_M - 2^m \cdot [e_m]_M$. Now all that remains is to open $[e']_M$ (which "hides" c using r_j) as well as $([e_0]_2, \ldots, [e_{m-1}]_2)$ and check that $e' \overset{?}{=} \sum_{i=0}^{m-1} 2^i \cdot e_i$.

Remark 1. *When $M = 2^k$, we can optimize Π_{Conv} by removing the conversion step 6(d), which uses `daBits`. Instead, we simply ignore the carry bit and set $e_m = 0$, then in step (f), we can compute e' by first opening $2^{k-m}(c+r)$, then divide this by 2^{k-m} to obtain $e' = c+r \mod 2^m$. This can then be compared with $\sum_{i=0}^{m-1} e_i$, as required.*

3.3 Proof of security

Due to space constraints, the full proof of security can be found in Appendix A. We summarize the proof below.

In order to prove the security of Π_{Conv}, we first observe that instead of letting $\mathcal{P}$ choose multiplication triples, we might equivalently model this by letting $\mathcal{P}$ specify circuits instead (that will be evaluated instead of the Ripple Carry Adder). Then, we define an abstraction of the protocol as a balls-and-bins type game, similar to [EGK+20], and analyze the success probability of an adversary in this game.$\mathcal{A}$ winning in this abstraction rather than in the protocol Π_{Conv}. We make this abstraction, as a straightforward analysis of the conversion protocol is rather complex. This is due to there being multiple ways for $\mathcal{A}$ to pass the check with a bad conversion tuple. The first is by corrupting K conversion tuples, then corrupting $K \cdot B$ `edaBits` and hoping that these end up in the right buckets, canceling out the errors in the conversion tuples. The second approach is to corrupt a set of `edaBits` and then guess the arrangement of these, thus yielding how many circuits $\mathcal{A}$ would have to corrupt in order to cancel out the errors of the conversion tuples. Furthermore, conversion tuples (and `edaBits`) may be corrupted in several ways. To avoid these issues, we describe an abstract security game which only provides a better chance for the adversary to win than the original protocol. In summary, we show the following:

Theorem 1. *The probability of Π_{Conv} not detecting at least one incorrect conversion tuple is upper bounded by 2^{-s} whenever $N \geq 2^{s/(B-1)}$ and $C = C' = B$ for bucket size $B \in \{3, 4, 5\}$.*

The proof can be found in Appendix A.1. The approach is similar to that of [EGK$^+$20], however in our case since the conversion tuples are now fixed to be one per bucket, we have not taken a random permutation across all edaBits and conversion tuples. Therefore, we need a different analysis to show that this restriction on the permutation still suffices.

Using this, in Appendix A.2 we then prove security of Π_{Conv}:

Theorem 2. *Let $N \geq 2^{s/(B-1)}$, $C = C' = C'' = B$ and $B \in \{3, 4, 5\}$ such that $\frac{s}{B-1} > B$, then protocol Π_{Conv} (Figure 5) UC-realises $\mathcal{F}_{\mathsf{Conv}}$ (Figure 3) in the $\mathcal{F}_{\mathsf{Dabit}}$-hybrid model. Specifically, no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most 2^{-s}.*

3.4 Faulty daBits

When working in $\mathbb{Z}_p$ (i.e. $M = p$), our previous protocol requires a source of `daBits`, namely, committed tuples $([b]_2, [b]_M)$, where b is a random bit. Generating consisting `daBits` requires verifying that $[b]_M$ indeed contains a bit, which is done with a potentially costly multiplication check by showing that $b(1 - b) = 0$. In this section, we optimise the protocol for the $\mathbb{Z}_p$ case by showing that Π_{Conv} remains secure even with potentially *faulty* `daBits`. More concretely, convertBit2A (which is part of the verification protocol) will use `daBits` which are only proven consistent modulo 2. This is much cheaper to achieve and avoids to check that b' is a bit.

Definition 3. *A* faulty *`daBit` is a pair $([b]_2, [b']_M)$ such that $b \equiv b' \bmod 2$, but not necessarily $b' \in \{0, 1\}$.*

In Step 6 of Π_{Conv} (Figure 5), `daBits` are used in convertBit2A (Figure 6) to transform the final carry bit $[e_m]_2$ from bitADDcarry (Figure 7) into $[e'_m]_M$ such that $e_m = e'_m$. We show that using a faulty `daBit` cannot help the adversary in passing the check with incorrect conversions. First, we observe that with a faulty `daBit`, the output becomes $e'_m = e_m + (-1)^{e_m} \cdot \delta$ where $\delta = (-1)^b \cdot (b' - b)$ where $(b' - b) > 1$ represents the difference (or error) between b' and b for the used `daBit` $([b]_2, [b']_M)$. As a result, $|e'_m| > 1$ where $|\cdot|$ denotes absolute value. This carry bit $[e'_m]_M$ is then used to remove the potential carry from $[c + r]_M$ by computing

$$[e']_M \leftarrow [c + r]_M - 2^m \cdot [e'_m]_M$$

in Step 6d of Π_{Conv}. However, when $e'_m \notin \{0, 1\}$ is multiplied with 2^m, we either subtract or add something much larger than what may be represented by m bits from $[c + r]_M$. Because this error is so large, it is impossible for the adversary to cancel this out with faulty multiplication triples or conversion tuples. Essentially, this holds because faulty triples only introduce a 1-bit error, and the result will always still be representable in m bits.

A full security analysis, showing that faulty `daBits` do not impact the security of Π_{Conv}, can be found in Appendix B.

Formally, we define an ideal functionality $\mathcal{F}_{\mathsf{FDabit}}$ (Figure20 in Appendix Appendix B) that encompasses this idea of two bits (b, b') such that $b' \equiv b \bmod 2$.

Creating Faulty `daBits` We now describe a revised `daBit` verification protocol Π_{FDabit} realizing $\mathcal{F}_{\mathsf{FDabit}}$, that, for a given `daBit` $([b]_2, [b']_M)$, only verifies that $b \equiv b' \bmod 2$. For this check, it suffices to compute random linear combinations in the two domains and then open the values, giving consistency modulo 2. We define the protocol Π_{FDabit} in Figure 8, and show the following statement in Appendix B:

Lemma 1. *Protocol Π_{FDabit} (Figure 8) UC-realizes $\mathcal{F}_{\mathsf{FDabit}}$ (Figure 20) except with probability 2^{-s+1}.*

We note that the main complexity of our faulty `daBit` check is just that of creating the $\gamma \cdot s$ auxiliary `daBits`. We assume the random bits e_i can be generated by letting the verifier pick a seed and then using some expansion function on both sides. Since $\gamma = s + \log N + 1$, the dominant communication cost — namely

Protocol Π_{FDabit}

Inputs N supposed faulty `daBits` $([b_i]_2, [b_i]_p)_{i\in[N]}$. Define $\gamma = s + \lceil \log_2(N+1) \rceil$ and require that $(N+1)\cdot 2^{\gamma+2} < (p-1)/2$ for a statistical security parameter s.

Protocol Perform the following check s times:

1. $\mathcal{P}$ samples γ random bits $\{[c_i]_p\}_{i\in[\gamma]}$. It additionally creates $[c_1]_2$. It does not prove consistency among $[c_1]_p$ and $[c_1]_2$.
2. $\mathcal{P}$ shows that $\{[c_i]_p\}_{i\in[\gamma]}$ are bits by showing that $\mathsf{CheckZero}([c_i]_p \cdot (1-[c_i]_p)) = (\mathsf{success})$ for $i \in [\gamma]$.
3. $\mathcal{V}$ generates N random bits e_i.
4. Let $[r]_2 \leftarrow [c_1]_2 + \bigoplus_{i=1}^{N} e_i \cdot [b_i]_2$
5. $r \leftarrow \mathsf{Open}([r]_2)$
6. Let $[r']_M \leftarrow [\sum_{i=1}^{N} e_i \cdot b_i]_p$
7. Let $\tau = \mathsf{Open}([r']_p + \sum_{j=1}^{\gamma} [c_i]_p \cdot 2^{i-1})$.
8. Check if $0 \le \tau < 2^\gamma$ and $r = \tau \bmod 2$. If not, abort.

Output $(\mathsf{success})$.

Figure 8: Our optimised consistency check for `daBits`, that no longer checks that $[b]_p$ is a bit

committing to $\gamma \cdot s$ `daBits` and multiplying to check that $c_i(1-c_i)$ is zero — amortizes away when N is large. This is in contrast to a secure `daBit` protocol, which would need incur these costs for every faulty `daBit`.

3.5 Complexity of Verifying Conversions

The amortized costs for verifying the correctness of a single conversion tuple $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ are given in Table 1, in terms of the amount of communication required, and preprocessed correlated OTs or VOLEs. Note that to simplify the table, we assume that $m \approx \log p$, and so count the cost of sending one $\mathbb{Z}_M$ element in the protocol as m bits. Also, in this analysis we ignore costs that are independent of the number of conversions being checked, such as the small number of checks in the faulty `daBit` protocol. In Appendix E, we give a more detailed breakdown of these costs, including complexities of the sub-protocols $\mathsf{bitADDcarry}$ and $\mathsf{convertBit2A}$.

Table 1: Costs of verifying a conversion between $\mathbb{Z}_2$ and $\mathbb{Z}_m$ (where either $m = p$ prime or $m = 2^k$) in form of the number of needed COTs and VOLEs, and the communication that is additionally required. The "basic" protocol variant uses `edaBits` directly, while the "optimized" rows include all of our novel optimizations. $m \le \lceil \log(p) \rceil$, k denotes the bitsize of the converted value, and B is the bucket size.

Protocol	Comm. in bits	#COTs	#VOLEs
basic, $\mathbb{Z}_p$, $\log(p) \le s$	$13Bm + 6m + B - 1$	$4Bm + 3m + B - 1$	$11B - 4$
basic, $\mathbb{Z}_p$, $\log(p) > s$	$10Bm + 6m + B - 1$	$4Bm + 3m + B - 1$	$8B - 4$
optimized, $\mathbb{Z}_p$	$6Bm + B$	$4Bm + B$	$2B$
optimized, $\mathbb{Z}_{2^k}$	$5Bm$	$4Bm$	B

The "basic" baseline comparison in Table 1 comes from a straightforward application of using `edaBits` for

ZK, similarly to [EGK+20]. Namely, this protocol would first generate consistent `edaBits` using [EGK+20], and then verify the conversion using a single binary addition circuit (similar to the the bucket-check in Figure 5, step 6). However, this requires doing the check with m *verified* multiplication triples (over $\mathbb{Z}_2$) and a single `daBit`, which in turn requires an additional verified multiplication (over $\mathbb{Z}_M$). To estimate these costs, we used the Wolverine [WYKW20] protocol for verifying AND gates at a cost of 7 bits per gate, and Mac'N'Cheese [BMRS20] for verifying triples in a larger field.

Since COTs and VOLEs can be obtained from pseudorandom correlation generators with very little communication [YWL+20, WYKW20], the remaining online communication dominates. Hence, our optimized protocol saves at least 50% communication. To give a concrete number, e.g. for the $\mathbb{Z}_p$ variant with $m = 32$, when verifying a batch of around a million triples and 40-bit statistical security, we can use bucket size $B = 3$, and the communication cost drops from 1442 to 579 bits, a reduction of around 60%.

Note that, as mentioned in Section 1.2, the "basic" approach could also be optimized by verifying these multiplications with the recent Quicksilver protocol [YSWW21]. This would bring the basic costs closer to the costs of our optimized protocols, while requiring non-black-box use of the information-theoretic MACs, unlike our more generic commit-and-prove protocol.

4 Truncation and Integer Comparison

In this section, we provide protocols for verifying integer truncation and comparison. With truncation, we mean that given integers l, m and two authenticated values x, x' of l and $l - m$ bits, we want to verify that x' corresponds to the upper $l - m$ bits of x, i.e. $x' = \lfloor \frac{x}{2^m} \rfloor$ over the integers. Integer comparison is then the problem of taking two authenticated integers and outputting 0 or 1 (authenticated) depending on which input is the largest. Both protocols take as input both the input and output of the function from the prover and then verify the correctness of the provided data.

We also describe a novel way of checking the length of an authenticated integer. We ask the prover to provide not only the authenticated ring element, but also its bit decomposition. By proving consistency of these two representations, the prover shows that the authenticated ring element can be represented by the provided bit decomposition of which we can check the length. The naïve way of achieving this would be using a protocol for integer comparison or a less-than circuit. However, both of these ways would require auxiliary consistent `edaBits` in addition to possibly other operations. Instead, we only have to verify that the input forms a consistent `edaBit` and therefore save anything beyond that.

We note that the integers in this section are signed in the interval $[-2^{l-1}, 2^{l-1})$, but the protocols are all defined over a modulus $M \geq 2^l$ where M is either a prime p or 2^k. Given an integer $\alpha \in [-2^{l-1}, 2^{l-1})$, this can be represented by a corresponding ring element in $\mathbb{Z}_M$.

4.1 Truncation

In Figure 9 we present a functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$ that takes a batch of commitments $[a_j]_M$ and their supposed truncations (by m^j bits) $[a'_j]_M$. The functionality ensures that the truncations are correct, namely, $a'_j = \lfloor \frac{a_j}{2^{m^j}} \rfloor$. Note that this functionality we realise is flexible, in that it can support a large batch of truncations, each of which may be of a different length.

We now construct a protocol for verifying truncations, which can securely realise $\mathcal{F}_{\mathsf{VerifyTrunc}}$ using just a *single* call to our batch conversion protocol, $\mathcal{F}_{\mathsf{Conv}}$, on a vector of tuples that is twice the length of the number of truncations. For the protocol, we will have that in addition to each input $[a]_M$, the prover also provides:

- the truncated value $[a_{tr}]_M$ of $[a]_M$ and its bit decomposition $([a_{tr}^0]_2, \ldots, [a_{tr}^{l-m-1}]_2)$
- the initial m bits of $[a]_M$; $[a']_M = [a \mod 2^m]_M$ as well as its bit decomposition $([a'_0]_2, \ldots, [a'_{m-1}]_2)$

Having access to $[a_{tr}]_M$ and $[a']_M$ allows the verifier then to check that $a = 2^m \cdot a_{tr} + a'$, which is sufficient to prove the claim.Observe that running Π_{Conv} on $[a_{tr}]_M$ and $([a_{tr}^0]_2, \ldots, [a_{tr}^{l-m-1}]_2)$ not only shows consistency

Functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$

The functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$ extends $\mathcal{F}^{2,M}_{\mathsf{ComZK}}$ with $\mathsf{VerifyTrunc}$ that verifies truncations of committed values from $\mathbb{Z}_M$. The function takes a set of IDs $\{(\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]}$ of elements $a^{\mathsf{id}^{0,j}}, a^{\mathsf{id}^{1,j}} \in \mathbb{Z}_M$ and a set of integers $\{m^j\}_{j\in[N]}$ such that $m^j \in [M]$ represents by how much $a^{\mathsf{id}^{0,j}}$ is truncated to reach $a^{\mathsf{id}^{1,j}}$ for $j \in [N]$. It is assumed that the underlying values of the id's have been Input prior to calling this method.

VerifyTrunc: Upon $\mathcal{P}$ and $\mathcal{V}$ inputting $(\mathsf{VerifyTrunc}, N, \{m^j, (\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]})$:

- Check that $a^{\mathsf{id}^{1,j}} = \lfloor \frac{a^{\mathsf{id}^{0,j}}}{2^{m^j}} \rfloor$, for each $j \in [N]$. If all checks pass, output $(\mathsf{success})$, otherwise abort.

Figure 9: Functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$ that verifies a truncation

between the binary and arithmetic representations, but also that $[a_{tr}]_M$ can be represented by $l-m$ or less bits (same goes for $[a']_M$ and its bit decomposition).

We first define an ideal functionality $\mathcal{F}_{\mathsf{CheckLength}}$ (Figure 10) that encapsulates this concept of using $\mathcal{F}_{\mathsf{Conv}}$ as a way of bounding the size of an authenticated value.

The protocol $\Pi_{\mathsf{CheckLength}}$ ensures that $[a]_M$ can be represented by m bits, as it proves consistency between the two representations of a. The security of this protocol directly follows from using $\mathcal{F}_{\mathsf{Conv}}$. The cost of the protocol also directly follows from the consistency check described in Figure 5.

We prove that $\Pi_{\mathsf{VerifyTrunc}}$ securely realises the functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$, in Appendix C. Note that $\Pi_{\mathsf{CheckLength}}$ and $\Pi_{\mathsf{VerifyTrunc}}$ do not utilise anything specific about M except $l \leq M$ and both work for $\mathbb{Z}_p$ and $\mathbb{Z}_{2^k}$.

Functionality $\mathcal{F}_{\mathsf{CheckLength}}$

This functionality extends $\mathcal{F}^{2,M}_{\mathsf{ComZK}}$ with the extra function $\mathsf{VerifyLength}$ that takes a set of IDs $\{\mathsf{id}^j\}_{j\in[N]}$ of elements $x^{\mathsf{id}^j} \in \mathbb{Z}_M$ and a set of integers $\{m^j\}_{j\in[N]}$ such that $m^j \in [M]$ represents the supposed lengths of the elements $\{x^{\mathsf{id}^j}\}_{j\in[N]}$. $\mathcal{F}_{\mathsf{CheckLength}}$ communicates with two parties $\mathcal{P}$, $\mathcal{V}$. It is assumed that the underlying values of the id's have been Input prior to calling this method.

VerifyLength: Upon $\mathcal{P}$ and $\mathcal{V}$ inputting $(\mathsf{VerifyLength}, N, \{m^j, \mathsf{id}^j\}_{j\in[N]})$:

- Check that x^{id^j} may be described by m^j bits for all $j \in [N]$. Output $(\mathsf{success})$ if so, otherwise abort.

Figure 10: Functionality to verify length of commitments

Theorem 3. *The protocol $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 12) UC-realizes $\mathcal{F}_{\mathsf{VerifyTrunc}}$ (Figure 9) in the $\mathcal{F}_{\mathsf{CheckLength}}$-hybrid model.*

4.2 Integer Comparison

We now discuss how to compare two signed, l-bit integers α and β. The way the protocol works is by having the prover (and verifier) compute $[\alpha]_M - [\beta]_M$ and have the prover compute the truncation of this which is only the most significant bit. Now we may run $\Pi_{\mathsf{VerifyTrunc}}$ on the truncation and use the truncation as the output of the comparison. We remark that, similarly to previous works in the MPC setting [Cd10, EGK$^+$20],

Protocol $\Pi_{\mathsf{CheckLength}}$

Input A set of tuples $\{[x^j]_M, m^j,\}_{j\in[N]}$ where $x^j \in [0, 2^l)$ and m^j defines the claimed bitlength of x^j.

Protocol

1. For each $j \in [N]$, $\mathcal{P}$ commits to $[x_0^j]_2, \ldots, [x_{m^j-1}^j]_2$.
2. Let $m = \max_j\{m^j\}$, and for $i = m^j, \ldots, m-1$, let $[x_i^j]_2$ denote a dummy commitment to zero (which can be easily obtained with $\mathsf{CheckZero}$).
3. Run $\mathcal{F}_{\mathsf{Conv}}$ on $\{([x_0^j]_2, \ldots, [x_{m-1}^j]_2, [x^j]_M)\}_{j\in[N]}$ and output what $\mathcal{F}_{\mathsf{Conv}}$ outputs.

Figure 11: Protocol $\Pi_{\mathsf{CheckLength}}$ that verifies that committed elements are bounded.

Protocol $\Pi_{\mathsf{VerifyTrunc}}$

Input A set of tuples $\{[a^j]_M, m^j, [a_{tr}^j]_M\}_{j\in[N]}$ where $a^j \in [0, 2^l)$, m^j defines the number of bits that has been truncated and $[a_{tr}]_M$ represents the supposed truncation.

Protocol

1. For each $j \in [N]$, $\mathcal{P}$ commits to the least-significant m bits of $[a^j]_M$, denoted as $[a']_M = [a^j \bmod 2^m]_M$.
2. The parties call $\mathcal{F}_{\mathsf{CheckLength}}$ with input $\{[a']_M, m^j\}_{j\in[N]} \cup \{[a_{tr}^j]_M, l - m^j\}_{j\in[N]}$.
3. For each j, let $[y]_M = [a^j]_M - (2^m \cdot [a_{tr}^j]_M + [a']_M)$ and run $\mathsf{CheckZero}([y]_M)$.

Abort if any of the checks fail. Otherwise output $(\mathsf{success})$.

Figure 12: Protocol to verify the truncation of an element from $\mathbb{Z}_M$

this gives the correct result as long as $\alpha, \beta \in [-2^{l-2}, 2^{l-2})$, so that $\alpha - \beta \in [-2^{l-1}, 2^{l-1})$, so this introduces a mild restriction on the range of values that can be supported.

5 Interactive Proofs over $\mathbb{Z}_{2^k}$

In this section, we provide the foundations for an interactive proof system that natively operates over $\mathbb{Z}_{2^k}$. First, we show how linearly homomorphic commitments for $\mathbb{Z}_{2^k}$ can be constructed from VOLE in Section 5.1. Then, in Section 5.2, we present two protocol variants which instantiate $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$, and prove their security in Section 5.3.

5.1 Linearly Homomorphic Commitments from Vector-OLE

To construct linearly homomorphic commitments over the ring $\mathbb{Z}_{2^k}$, we use a variant of the information-theoretic MAC scheme from SPD$\mathbb{Z}_{2^k}$ [CDE+18]: Let s be a statistical security parameter. To authenticate a value $x \in \mathbb{Z}_{2^k}$ known to $\mathcal{P}$ towards $\mathcal{V}$ (denoted as $[x]$), we choose the MAC keys $\Delta \in_R \mathbb{Z}_{2^s}$ and $K[x] \in_R \mathbb{Z}_{2^{k+s}}$, and compute the MAC tag as

$$M[x] := \Delta \cdot \tilde{x} + K[x] \in \mathbb{Z}_{2^{k+s}} \tag{1}$$

where $x = \tilde{x} \bmod 2^k$, i.e. $\tilde{x}$ is a representative of the corresponding congruence class of integers modulo 2^k. Then $\mathcal{P}$ gets $\tilde{x}$ and $M[x]$, whereas $\mathcal{V}$ receives Δ and $K[x]$.

Initially $\tilde{x}$ may be chosen as $\tilde{x} = x \in \{0, \dots, 2^k - 1\}$. Applying the arithmetic operations described below can result in larger values though, which do not get reduced modulo 2^k because all computation happens modulo 2^{k+s}. For a commitment $[x]$ we always use $\tilde{x}$ to denote the representative hold by $\mathcal{P}$.

This MAC schemes allows us to locally compute affine combinations: E.g. for $[z] \leftarrow a \cdot [x] + [y] + b$ with public $a, b \in \mathbb{Z}_{2^k}$, the parties compute $\tilde{z} \leftarrow a \cdot \tilde{x} + \tilde{y} + b$ and $M[z] \leftarrow a \cdot M[x] + M[y]$, as well as $K[z] \leftarrow a \cdot K[x] + K[y] - \Delta \cdot b$. Then we have

$$\begin{aligned} M[z] &\equiv_{k+s} a \cdot M[x] + M[y] \\ &\equiv_{k+s} a \cdot (\Delta \cdot \tilde{x} + K[x]) + (\Delta \cdot \tilde{y} + K[y]) \\ &\equiv_{k+s} \Delta \cdot (a \cdot \tilde{x} + \tilde{y}) + (a \cdot K[x] + K[y]) \\ &\equiv_{k+s} \Delta \cdot (a \cdot \tilde{x} + \tilde{y} + b) + (a \cdot K[x] + K[y] - \Delta \cdot b) \\ &\equiv_{k+s} \Delta \cdot \tilde{z} + K[z]. \end{aligned}$$

While we can initially set $\tilde{x} = x$, a result of a computation (here $\tilde{z}$) might be larger than $2^k - 1$, but for the computation we only care about the lower k bits of $\tilde{z}$ (denoted as z).

As in SPD$\mathbb{Z}_{2^k}$, the MACs are obtained using vector OLE over rings. We describe the protocols in the $\mathcal{F}^{s,r}_{\mathsf{vole2k}}$-hybrid model (cf. Figure 13). This functionality can be instantiated as in [CDE+18], e.g. from the oblivious transfer protocol of [Sch18]. To open a commitment $[x]$, first the upper s bits of $\tilde{x}$ need to be randomized, by computing $[z] \leftarrow [x] + 2^k \cdot [r]$ with random $\tilde{r} \in_R \mathbb{Z}_{2^{k+s}}$. Then, $\tilde{z}$ and $M[z]$ are published and the MAC equation (Equation (1)) is verified. Following [CDE+18], we implement more efficient batched checks based on random linear combinations. These are implemented in protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 14 & 15).

Vector Linear Oblivious Evaluation for $\mathbb{Z}_{2^k}$: $\mathcal{F}^{s,r}_{\mathsf{vole2k}}$

Init This method needs to be the first one called by the parties. On input (Init) from both parties the functionality

1. If $\mathcal{V}$ is honest, it samples $\Delta \in_R \mathbb{Z}_{2^s}$ and sends Δ to $\mathcal{V}$.
2. If $\mathcal{V}$ is corrupt, it receives $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{S}$.
3. Δ is then stored by the functionality.

All further Input queries are ignored.

Extend On input (Extend) from both parties the functionality proceeds as follows:

1. If both parties are honest, sample $x, K[x] \in_R \mathbb{Z}_{2^r}$ and compute $M[x] \leftarrow \Delta \cdot x + K[x] \in_R \mathbb{Z}_{2^r}$.
2. If $\mathcal{V}$ is corrupted, it receives $K[x] \in \mathbb{Z}_{2^r}$ from $\mathcal{S}$ instead.
3. If $\mathcal{P}$ is corrupted, it receives $x, M[x] \in \mathbb{Z}_{2^r}$ from $\mathcal{S}$ instead, and computes $K[x] \leftarrow M[x] - \Delta \cdot x \in \mathbb{Z}_{2^r}$.
4. $(x, M[x])$ is sent to $\mathcal{P}$ and $K[x]$ is sent to $\mathcal{V}$.

Figure 13: Ideal functionality for Vector-OLE with key size s and message size r. The functionality is based on $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ from Wolverine [WYKW20, Fig. 2].

5.2 Instantiation of $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$

In this section, we present two protocols $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ and $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ which instantiate the $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ functionality (Figure 1). These are adaptions of the Wolverine [WYKW20] and Mac'n'Cheese [BMRS20] protocols to the $\mathbb{Z}_{2^k}$ setting and differ mainly in the implementation of the $\mathsf{CheckMult}$ method.

$\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ (Figures 14 & 15) adapts the bucketing approach from Wolverine [WYKW20]: Let $C, B \in \mathbb{N}$ be the parameters of the bucketing scheme. To check that a collection of triples $([a_i], [b_i], [c_i])_{i=1}^n$ satisfy a multiplicative relation, i.e. $a_i \cdot b_i = c_i$ for $i = 1, \ldots, n$, the prover creates a set of $\ell := n \cdot B + C$ unchecked multiplication triples of commitments. After randomly permuting the ℓ triples according to the choice of the verifier, C triples are opened and checked by the verifier. The remaining nB triples are evenly distributed into n buckets. Then, each multiplication $(a_i \cdot b_i \stackrel{?}{=} c_i)$ is verified with the B triples of the corresponding bucket with a variant of Beaver's multiplication trick [Bea92]. For the check to pass despite an invalid multiplication $a_i \cdot b_i \neq c_i$, the adversary needs to corrupt exactly those triples that end up in the corresponding bucket for that multiplication.

For $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ (Figure 16), we have adapted the multiplication check of Mac'n'Cheese [BMRS20], which is similar to the Wolverine [WYKW20] optimization for large fields and SPDZ-style [DPSZ12] sacrificing of multiplication triples. The soundness of this type of check is based on the difficulty of finding a solution to a randomized equation. If a multiplicative relation does not hold, the adversary needs to guess a random field element in order to pass. Thus the original scheme needs a large field to be sound. In the $\mathbb{Z}_{2^k}$-setting, there are multiple obstacles that we have to overcome. First, we would like to also support small values of k (e.g. $k = 8$ or 16). Simultaneously, we also have to deal with zero divisors (which complicate the check) which were no issue in the field setting. Moreover, even though the commitment scheme (see Section 5.1) uses the larger ring $\mathbb{Z}_{2^{k+s}}$ it only authenticates the lower k bits of $\tilde{x}$ and cannot prevent modifications of the upper bits, which might lead to additional problems. We overcome these challenges by further increasing the ring size from $\mathbb{Z}_{2^{k+s}}$ to $\mathbb{Z}_{2^{k+2s}}$, so that the commitment scheme provides authenticity of values modulo 2^{k+s}. We use the additional s bits to avoid overflows when checking correctness of the multiplicative relations modulo 2^k with an s bit random challenge. Increasing the ring leads to larger storage requirements – the values $\tilde{x}, M[x], K[x]$ now require $k + 2s$ bits. It has, however, essentially no influence on the communication costs of $\mathsf{CheckZero}$ and Open, since we need to mask the additional s bits only once, independently of the number of checked commitments.

5.3 Proofs of Security

In this section we formally state the security guarantees of our protocols. and give an overview of the corresponding proofs. Due to space limits, most of the complete proofs are given in Appendix F.

5.3.1 Proof of $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$

The formal statement of security is given in the following theorem:

Theorem 4. *The protocol $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ (Figures 14 & 15) securely realizes the functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$: No environment can distinguish the real execution from a simulated one except with probability $(q_{\mathsf{cz}} + q_{\mathsf{cm}}) \cdot 2^{-s+\log(s+1)} + q_{\mathsf{cm}} \cdot \binom{nB+C}{B}^{-1}$, where q_{cz} is the sum of calls to* $\mathsf{CheckZero}$ *and* Open*, and q_{cm} the number of calls to* $\mathsf{CheckMult}$.

We prove the theorem in the UC model by constructing a simulator that generates a view indistinguishable to that in a real protocol execution. In the case of a corrupted verifier, the simulation is perfect. For a corrupted prover, the distinguishing advantage depends on thes oundness properties of the $\mathsf{CheckZero}$ and $\mathsf{CheckMult}$ protocols in $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$. These are stated in the following two lemmata. The full proof of Theorem 4 is given in Appendix F.3.

Lemma 2. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* $\mathsf{CheckZero}$ *protocol of $\Pi_{\mathsf{com-a}}^{\mathbb{Z}_{2^k}}$ with commitments $[x_1], \ldots, [x_n]$ and $x_i \not\equiv_k 0$ for some $i \in \{1, \ldots, n\}$ then $\mathcal{V}$ outputs* $(\mathsf{success})$ *with probability at most $\varepsilon_{\mathsf{cz}} := 2^{-s+\log(s+1)}$.*

Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Part 1)

Each party can abort the protocol by sending the message (abort) to the other party and terminating the execution.

Init For (Init), the parties send (Init) to $\mathcal{F}^{s,k+s}_{\mathsf{vole2k}}$. $\mathcal{V}$ receives its global MAC key $\Delta \in \mathbb{Z}_{2^s}$.

Random For (Random), the parties send (Extend) to $\mathcal{F}^{s,k+s}_{\mathsf{vole2k}}$ so that $\mathcal{P}$ receives $M[r], r \in \mathbb{Z}_{2^{k+s}}$ and $\mathcal{V}$ receives $K[r] \in \mathbb{Z}_{2^{k+s}}$ so that $M[r] = \Delta \cdot r + K[r]$ holds. This is denoted as $[r]$.

Affine Combination For $[z] \leftarrow \alpha_0 + \sum_{i=1}^n \alpha_i \cdot [x_i]$, the parties locally set

- $\tilde{z} \leftarrow \alpha_0 + \sum_{i=1}^n \alpha_i \cdot \tilde{x}_i$ (by $\mathcal{P}$),
- $M[z] \leftarrow \sum_{i=1}^n \alpha_i \cdot M[x_i]$ (by $\mathcal{P}$),
- $K[z] \leftarrow -\Delta \cdot \alpha_0 + \sum_{i=1}^n \alpha_i \cdot K[x_i]$ (by $\mathcal{V}$).

CheckZero For $(\mathsf{CheckZero}, [x_1], \dots, [x_n])$, the parties proceed as follows:

1. If one of the x_i is not equal to 0, then $\mathcal{P}$ aborts.
2. They run $[r] \leftarrow \mathsf{Random}()$.
3. $\mathcal{V}$ samples $\chi_1, \dots, \chi_n \in_R \mathbb{Z}_{2^s}$ and sends them to $\mathcal{P}$.
4. Let $p_i := (\tilde{x}_i - x_i)/2^k$ denote the upper s bits of $\tilde{x}_i$. $\mathcal{P}$ computes $p \leftarrow \tilde{r} + \sum_{i=1}^n \chi_i \cdot p_i \in \mathbb{Z}_{2^s}$ and $m \leftarrow \sum_{i=1}^n \chi_i \cdot M[x_i] + 2^k \cdot M[r] \in \mathbb{Z}_{2^{k+s}}$, and sends both to $\mathcal{V}$.
5. Finally, $\mathcal{V}$ checks $m \overset{?}{\equiv}_{k+s} 2^k \cdot \Delta \cdot p + \sum_{i=1}^n \chi_i \cdot K[x_i] + 2^k \cdot K[r]$, and outputs $(\mathsf{success})$ if the equality holds and aborts otherwise

Input For (Input, x), where $x \in \mathbb{Z}_{2^k}$ is known by $\mathcal{P}$, the parties first run $[r] \leftarrow \mathsf{Random}()$. Then $\mathcal{P}$ sends $\delta := x - r \bmod 2^k$ to $\mathcal{V}$, and they compute $[x] \leftarrow [r] + \delta$.

Open For $(\mathsf{Open}, [x_1], \dots, [x_n])$, $\mathcal{P}$ sends $x_1, \dots, x_n$ to $\mathcal{V}$, and they compute $[z_i] \leftarrow [x_i] - x_i$ for $i = 1, \dots, n$, followed by $\mathsf{CheckZero}([z_1], \dots, [z_n])$. The result of the latter is returned.

Figure 14: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ using a Wolverine-like [WYKW20] multiplication check.

Since the CheckZero protocol is based on the MAC check from [CDE+18], the proof of Lemma 2 is similar. It is given in Appendix F.1.

Lemma 3. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckMult *protocol of $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{com-a}}$ with parameters $B, C \in \mathbb{N}$ such that $C \geq B$ and inputs $([a_i], [b_i], [c_i])_{i=1}^n$ and there exists an index $1 \leq i \leq n$ such that $a_i \cdot b_i \not\equiv_k c_i$ then $\mathcal{V}$ outputs* (success) *with probability at most $\varepsilon_{\mathsf{cm}} + \varepsilon_{\mathsf{cz}}$ with $\varepsilon_{\mathsf{cm}} := \binom{nB+C}{B}^{-1}$, and $\varepsilon_{\mathsf{cz}}$ the soundness error of* CheckZero *given in Lemma 2.*

The CheckMult protocol is based on the corresponding check from Wolverine [WYKW20], and the same analysis also applies to the $\mathbb{Z}_{2^k}$ case. The proof of Lemma 3 can be found in Appendix F.2.

5.3.2 Proof of $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$

The formal statement of security is given in the following theorem:

Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\textsf{ComZK-a}}$ (Part 2)

MultiplicationCheck Let $B, C \in \mathbb{N}$ be parameters of the protocol. On input $(\textsf{CheckMult}, ([a_i], [b_i], [c_i])_{i=1}^n)$ the parties proceed as follows:

1. $\mathcal{P}$ aborts if $a_i \cdot b_i \neq c_i \pmod{2^k}$ for some $i = 1, \dots, n$.
2. Let $\ell := n \cdot B + C$, and initialize $\textsf{lst} \leftarrow \emptyset$
3. They compute $([x_i], [y_i])_{i=1}^{\ell} \leftarrow \textsf{Random}()$ so that $\mathcal{P}$ receives $(x_i, y_i)_{i=1}^{\ell}$.
4. $\mathcal{P}$ computes $z_i \leftarrow x_i \cdot y_i$ for $i = 1, \dots, \ell$, and they run $([z_i])_{i=1}^{\ell} \leftarrow \textsf{Input}((z_i)_{i=1}^{\ell})$.
5. $\mathcal{V}$ samples a permutation $\pi \in_R S_\ell$ and sends it to $\mathcal{P}$.
6. They run $(x_{\pi(i)}, y_{\pi(i)}, z_{\pi(i)})_{i=1}^{C} \leftarrow$ $\textsf{Open}(([x_{\pi(i)}], [y_{\pi(i)}], [z_{\pi(i)}])_{i=1}^{C}, \textsf{lst})$.
7. $\mathcal{V}$ checks if $x_{\pi(i)} \cdot y_{\pi(i)} = z_{\pi(i)}$ for $i = 1, \dots, C$, and aborts otherwise.
8. For each (a_j, b_j, c_j) with $j = 1, \dots, n$ and for each $(x_{\pi(k)}, y_{\pi(k)}, z_{\pi(k)})$ with $k = C + (j - 1) \cdot B + 1, \dots, C + j \cdot B$, they compute
   (a) $d \leftarrow \textsf{Open}([a_j] - [x_{\pi(k)}], \textsf{lst})$ and $e \leftarrow \textsf{Open}([b_j] - [y_{\pi(k)}], \textsf{lst})$ and
   (b) $[w_k] \leftarrow [z_{\pi(k)}] - [c_j] + e \cdot [x_{\pi(k)}] + d \cdot [y_{\pi(k)}] + d \cdot e$
9. Finally, they run $(\textsf{CheckZero}, \textsf{lst}, ([w_k])_{k=C+1}^{\ell})$. If successful and the check in Step 7 also passed, $\mathcal{V}$ outputs $(\textsf{success})$ and aborts otherwise.

Figure 15: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\textsf{ComZK-a}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\textsf{ComZK}}$ using a Wolverine-like [WYKW20] multiplication check.

Theorem 5. *The protocol $\Pi^{\mathbb{Z}_{2^k}}_{\textsf{ComZK-b}}$ (Figure 16) securely realizes the functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\textsf{ComZK}}$: No environment can distinguish the real execution from a simulated one except with probability $(q_{\textsf{cz}} + q_{\textsf{cm}}) \cdot 2^{-s+\log(s+1)} + q_{\textsf{cm}} \cdot 2^{-s}$, where $q_{\textsf{cz}}$ is the sum of calls to* CheckZero *and* Open*, and $q_{\textsf{cm}}$ the number of calls to* CheckMult*.*

The proof of Theorem 5 is given in Appendix F.4. Except for the simulation of CheckMult, it is largely similar to the proof of Theorem 4. Again, we first prove a lemma about the soundness error of the CheckMult operation, that we use to show indistinguishability of our simulation. The proof is included here, since it shows why the adaption of the SPDZ-style sacrificing check from large fields to the $\mathbb{Z}_{2^k}$ setting is secure.

Lemma 4. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckMult *protocol of $\Pi^{\mathbb{Z}_{2^k}}_{\textsf{ComZK-b}}$ with inputs $([a_i], [b_i], [c_i])_{i=1}^n$ such that there exists an index $1 \leq i \leq n$ such that $a_i \cdot b_i \not\equiv_k c_i$, then $\mathcal{V}$ outputs* (success) *with probability at most $\varepsilon'_{\textsf{cm}} + \varepsilon_{\textsf{cz}}$ with $\varepsilon'_{\textsf{cm}} := 2^{-s}$, and $\varepsilon_{\textsf{cz}}$ the soundness error of* CheckZero *given in Lemma 2.*

Proof. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the CheckMult protocol with inputs as described in the lemma.

Since CheckZero$'$ is a variant of CheckZero from $\Pi^{\mathbb{Z}_{2^k}}_{\textsf{ComZK-a}}$ for the larger message space $\mathbb{Z}_{2^{k+s}}$, we can apply Lemma 2 again: Hence, a $\mathcal{P}^*$ that tries to cheat during CheckZero$'$ is detected by $\mathcal{V}$ except with probability $\varepsilon_{\textsf{cz}}$.

Now assume this does not happen, all the zero checks are correct, and $\mathcal{V}$ accepts. Let i be an index of an invalid triple such $a_i \cdot b_i \not\equiv_k c_i$. Then, $\mathcal{P}$ has chosen $z_i \in \mathbb{Z}_{s^{k+s}}$ such that

$$\begin{aligned} 0 &\equiv_{k+s} \eta \cdot c_i - z_i - \varepsilon_i \cdot b_i \equiv_{k+s} \eta \cdot c_i - z_i - \eta \cdot a_i \cdot b_i + x_i \cdot b_i \\ \iff \qquad & z_i - x_i \cdot b_i \equiv_{k+s} \eta \cdot (c_i - a_i \cdot b_i). \end{aligned}$$

Let $v \in \mathbb{N}$ be maximal such that 2^v divides $c_i - a_i \cdot b_i$. Since (a_i, b_i, c_i) is an invalid triple modulo 2^k, it is $v < k$. Now we divide both sides of the equation by 2^v while also reducing the modulus to obtain:

$$(z_i - x_i \cdot b_i)/2^v \equiv_{k+s-v} \eta \cdot (c_i - a_i \cdot b_i)/2^v$$

Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$

Much of the protocol is identical to $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 14 and 15) with the exception that the MACs are now computed in the larger ring $\mathbb{Z}_{2^{k+2s}}$. **Init**, **Random**, **Affine Combination**, **Input** and **Open** work exactly as in $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$, although using $\mathcal{F}^{s,k+2s}_{\mathsf{vole2k}}$.

CheckZero For $(\mathsf{CheckZero}, [x_1], \dots, [x_n])$, the parties proceed as follows:

1. If one of the x_i is not equal to 0, then $\mathcal{P}$ aborts.
2. They run $[r] \leftarrow \mathsf{Random}()$.
3. $\mathcal{V}$ samples $\chi_1, \dots, \chi_n \in_R \mathbb{Z}_{2^s}$ and sends them to $\mathcal{P}$.
4. Let $p_i := (\tilde{x}_i - x_i)/2^k$ denote the upper $2s$ bits of $\tilde{x}_i$. $\mathcal{P}$ computes $p \leftarrow r + \sum_{i=1}^n \chi_i \cdot p_i \in \mathbb{Z}_{2^{2s}}$ and $m \leftarrow \sum_{i=1}^n \chi_i \cdot M[x_i] + 2^k \cdot M[r] \in \mathbb{Z}_{2^{k+2s}}$, and sends both to $\mathcal{V}$.
5. Finally, $\mathcal{V}$ checks $m \overset{?}{\equiv}_{k+2s} 2^k \cdot \Delta \cdot p + \sum_{i=1}^n \chi_i \cdot K[x_i] + 2^k \cdot K[r]$, and outputs $(\mathsf{success})$ if the equality holds and aborts otherwise.

CheckZero' $\mathsf{CheckZero}'$ denotes a variant of the above which checks that $\tilde{x}_i = 0 \pmod{2^{k+s}}$, and is only used in the multiplication check below. The difference is that only the upper s bits of the $\tilde{x}_i$ are hidden by p (now from $\mathbb{Z}_{2^s}$) instead of the upper $2s$ bits. The macro $\mathsf{Open}'([x], \mathsf{lst})$ is similarly an adaption revealing the lower $k+s$ bits and using $\mathsf{CheckZero}'$.

MultiplicationCheck The parties proceed on input $(\mathsf{CheckMult}, ([a_i],[b_i],[c_i])_{i=1}^n)$ as follows:

1. $\mathcal{P}$ aborts if $a_i \cdot b_i \neq c_i \pmod{2^k}$ for some $i = 1, \dots, n$.
2. Let $\mathsf{lst} := \emptyset$.
3. Generate $([x_i])_{i=1}^n \leftarrow \mathsf{Random}()$ followed by $[z_i] \leftarrow \mathsf{Input}(x_i \cdot b_i)$ for $i = 1, \dots, n$.
4. $\mathcal{V}$ sends a random value $\eta \in_R \mathbb{Z}_{2^s}$ to $\mathcal{P}$.
5. Compute $\varepsilon_i \leftarrow \mathsf{Open}'(\eta \cdot [a_i] - [x_i], \mathsf{lst})$ for $i = 1, \dots, n$.
6. Run $\mathsf{CheckZero}'((\eta \cdot [c_i] - [z_i] - \varepsilon_i \cdot [b_i])_{i=1}^n, \mathsf{lst})$. If successful, $\mathcal{V}$ returns $(\mathsf{success})$, otherwise abort.

Figure 16: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ using a Mac'n'Cheese-style [BMRS20] multiplication check.

Since $(c_i - a_i \cdot b_i)/2^v$ is odd, it is invertible modulo 2^{k+s-v} and we can move it to the other side, getting

$$(z_i - x_i \cdot b_i)/2^v \cdot ((c_i - a_i \cdot b_i)/2^v)^{-1} \equiv_{k+s-v} \eta.$$

Since $k > v$, we have $k + s - v > s$, and the prover would have guessed all s bits of $\eta \in \mathbb{Z}_{s^s}$ which happens only with probability 2^{-s}. Therefore, by the union bound, $\mathcal{P}^*$ can make $\mathcal{V}$ output $(\mathsf{success})$ with probability at most $\varepsilon_{\mathsf{cz}} + 2^{-s}$. $\square$

5.3.3 Communication Costs

In the protocols $\mathcal{V}$ samples random coefficients χ_i in $\mathsf{CheckZero}$ and a permutation π in $\mathsf{CheckMult}$ of $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ and sends these to $\mathcal{P}$. To reduce the communication costs, $\mathcal{V}$ can send a random seed instead, which both parties expand with a PRG to derive the desired random values. In this way, $\mathcal{V}$ needs to transfer only λ bits (for a computational security parameter λ) instead of $n \cdot s$ bits for $\mathsf{CheckZero}$ and $\log_2(n \cdot B + c)!$ bits for $\mathsf{CheckMult}$.

Table 2: Comparison of amortized communication costs in bits per commitment/triple to verify. k is the size of the modulus, s depends on the statistical security parameter, B is the bucket size used in $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$.

Protocol	CheckZero	Open	CheckMult
$\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$	0	k	$3Bk$ and $3B$ VOLEs
$\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$	0	k	$2 \cdot (k + s)$ and 2 VOLEs

The amortized communication costs per checked commitment and multiplication triple of both protocols are given in Table 2.

Acknowledgements

This material is based upon work supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC), the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM), and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085. 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 the Defense Advanced Research Projects Agency (DARPA). Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 223–254. Springer, Heidelberg, August 2017.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[BCG+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[BCG+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BL17] Carsten Baum and Vadim Lyubashevsky. Simple amortized proofs of shortness for linear relations over polynomial rings. Cryptology ePrint Archive, Report 2017/759, 2017. `https://eprint.iacr.org/2017/759`.

[BMRS20] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. `https://eprint.iacr.org/2020/1410`.

[Cd10] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

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

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019.

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446, 2020. `https://eprint.iacr.org/2020/1446`.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[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.

[EGL82] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 205–210. Plenum Press, New York, USA, 1982.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[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.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[IR89] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st ACM STOC*, pages 44–61. ACM Press, May 1989.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[MRVW21] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. Financial Crypto 2021, 2021.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

[Sch18] Peter Scholl. Extending oblivious transfer with low communication via key-homomorphic PRFs. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 554–583. Springer, Heidelberg, March 2018.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1055–1072. ACM Press, November 2019.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Report 2021/076, 2021. `https://eprint.iacr.org/2021/076`.

[YWL$^+$20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1607–1626. ACM Press, November 2020.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

A Proof of Security of the Basic Conversion Protocol

A.0.1 The RealGame

In order to prove the security of Π_{Conv}, we define an abstract game **RealGame** (Figure 17). In this abstraction, the prover (or $\mathcal{A}$) will pick additively tampered binary circuits directly, rather than individual multiplication triples. Apart from this change, the check run on each Conversion Tuple within each bucket is the same (step 6). We define all elements used to verify the consistency as *checking tuples.* These checking tuples contain an `edaBit`, a `daBit` and a potentially tampered circuit.

The RealGame

1. $\mathcal{A}$ prepares $N + (NB + C)$ authenticated `edaBits` (representing the N conversion tuples and $NB + C$ `edaBits` required to run the check), $\{([r_0^j]_2, \ldots, [r_{m-1}^j]_2, [r^j]_M)\}_{j\in[N+NB+C]}$, $NB + C'$ potentially tampered circuits $\{C^{j,*}\}_{j\in[NB+C']}$ and lastly NB `daBits` $\{([b^j]_2, [b^j]_M)\}_{j\in[NB]}$. These are all send to the challenger.
2. The challenger shuffles the N `edaBits` representing conversion tuples, then shuffles the remaining as well as the circuits using 3 permutations.
3. The challenger opens C checking `edaBits` and C' of the circuits. If any of the `edaBits` or circuits are inconsistent, terminate.
4. The challenger pairs up the remaining NB circuits with the NB `daBits` according to their permutations.
5. The challenger lets the shuffled list of the N `edaBits` be the first `edaBit` of each bucket before pairing up the remaining NB checking `edaBits` and NB circuits with the buckets.
6. Within each of the buckets, for every pair of `edaBits` $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)$ (where this is the first of the bucket) and $([s_0]_2, \ldots, [s_{m-1}]_2, [s]_M)$ take the next circuit C^* and compute $(c_0, \ldots, c_m) \leftarrow C^*(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1})$. Compute $c \leftarrow \sum_{i=0}^{m-1} c_i 2^i$ and check $r + s - 2^m \cdot c_m \stackrel{?}{=} c$.

$\mathcal{A}$ wins if all of the checks pass and there is a least one inconsistent `edaBit` as the top element of a bucket.

Figure 17: An abstraction of the `cutNchoose` protocol used to verify conversion tuples

This game models the conversion verification protocol closely, but is also difficult to analyze. We therefore make some simplifying assumptions about this game and arrive at the **SimpleGame** that we present in the next section.

A.0.2 The SimpleGame

We define an additional abstraction to the **RealGame**. This **SimpleGame** is even simpler in the sense that it no longer considers `edaBits`, triples or `daBits`. We argue that this simplified game **SimpleGame** models the **RealGame**, before analyzing the probability of success for the **SimpleGame**.

Within the **SimpleGame** `edaBits` are transformed into balls in such a way that a good `edaBit` is a clear ball (○) and bad (corrupt) `edaBits` are shades of gray balls (I.e. ● or ●) where each shade defines a different kind of corruption. Likewise, a good circuit is a clear triangle (△) and bad circuits are gray triangles (▲). A bad ball (or triangle) is bad in the sense that it helps the adversary win the game. Everyone is given access to the public function f that takes two balls and a triangle and outputs 0 or 1. This function f is

The SimpleGame

1. $\mathcal{A}$ prepares $N + (NB + C)$ balls and corrupts b of the $NB + C$ balls. These are all sent to the challenger.
2. The challenger opens C of the $NB + C$ balls at random and checks whether all C are good. If any of these balls are bad, then terminate.
3. The challenger shuffles the initial N and the remaining NB balls individually and then associates the initial N balls individually with N buckets. The challenger then randomly assigns the remaining NB balls to the N buckets which each have capacity B. The arrangement of balls is send to $\mathcal{A}$.
4. $\mathcal{A}$ prepares $NB + C'$ triangles and corrupts t of them. These are all sent to the challenger.
5. The challenger opens C' of the triangles at random and checks whether all C' are good. If any of these triangles are bad, then terminate.
6. The challenger shuffles the remaining NB triangles and randomly assigns these to the N buckets.
7. The challenger runs the `bucketcheck` procedure (Figure 19).
8. If `bucketcheck` returns 1, the challenger accepts the first ball of each bucket, otherwise terminate.

$\mathcal{A}$ wins if the protocol has not terminated at this point and at least one bad ball is accepted by the challenger.

Figure 18: A game working as a further abstraction of the **RealGame**

isomorphic to the winning condition in step 6 of **RealGame** and is modelled by the bucket check procedure shown in Figure 19. Finally, the adversary wins if `bucketcheck` does not abort, meaning that $\mathcal{A}$ passed all the checks, and there is at least one bad ball in the output.

Simplified bucket check for conversion tuples

Input

N buckets and a function f. Each bucket contains $B + 1$ balls $\{x_1, \ldots, x_B\}$ and B triangles $\{y_1, \ldots, y_B\}$.

Protocol

For each bucket, run the following check:

1. Check the configuration of $[x_1, x_i | y_{i-1}], \forall i \in [2, B]$.
   - If $[x_1, x_i | y_{i-1}] \in \{$[○,○|▲], [○,●|△], [●,○|△]$\}$ return `reject`.
   - If $[x_1, x_i | y_{i-1}] \in$ [●,●|△] and f(●,●,△) $= 0$ return `reject`.
   - If $[x_1, x_i | y_{i-1}] \in$ [●,●|▲] and f(●,●,▲) $= 0$ return `reject`.
2. Otherwise return `accept`.

If all checks returns `accept`, then output 1. Otherwise output 0.

Figure 19: A bucket check procedure used to check consistency of conversion tuples in the **SimpleGame**

In each check within the buckets, two balls are placed as well as one triangle. If the size of the buckets $B = 3$, then one bucket contains four balls $[B1, B2, B3, B4]$ and three triangles $[T1, T2, T3]$. The bucketcheck procedure then checks all of the configurations $[B1, B2, T1]$, $[B1, B3, T2]$, $[B1, B4, T3]$ and check if any of these configurations are {[○, ○, ▲}, {[○, ●, △}, {[●, ○, △} in which case the check fails and terminate. When there are two bad balls and one triangle (good or bad) however, whether or not to terminate depends on the type of the bad balls. This means we consider bad balls to be of different types (i.e. the prover provide conversions tuples and `edaBits` with different types of corruption) and we distinguish these with different color shades. As a result of this, the procedure might terminate if the configuration matches [●, ●, ▲] and in other cases it terminates due to [⚫, ●, △]. As the adversary will need to match the balls with triangles, an isomorphic argument can be made using different shadings for the triangles.

For clarity, we list the different advantageous (for the adversary) combinations of balls and triangles in Table 3. If the configurations within the buckets match those of the first three entries of Table 3 then bucketcheck will not terminate. If any match the penultimate entry or the final entry, then bucketcheck terminates if the output of f is 0.

Table 3: Favorable combinations of balls and triangles for the adversary

Circles		Triangles
○	○	△
○	●	▲
●	○	▲
●	●	△ / ▲
●	⚫	▲ / △

We now show that **RealGame** can be modeled as **SimpleGame** such that if **SimpleGame** is secure, then so is **RealGame**.

Lemma 5. *Security against all adversaries in **SimpleGame** implies security against all adversaries in **RealGame**.*

Proof. (Sketch.) We argue the security of our revised **RealGame** by showing that if there exists an efficient adversary $\mathcal{B}$ that wins **RealGame** with non-negligible probability, then there exists an efficient adversary $\mathcal{A}$ against **SimpleGame** that wins with non-negligible probability. $\mathcal{A}$ will simulate the **RealGame** challenger and then use $\mathcal{B}$ to win **SimpleGame**.

Keep in mind that the point of **SimpleGame** is to mix circles and triangles where a gray triangle (▲) corresponds to a faulty binary addition circuit and an empty triangle (△) represents a regular binary addition circuit, but both of these representations are purely semantics.

As mentioned, the adversary $\mathcal{A}$ simulates the challenger of **RealGame** and then uses $\mathcal{B}$ to win **SimpleGame**. $\mathcal{B}$ sends a batch of `edaBits` and a set of circuits (and `daBits`) to $\mathcal{A}$. $\mathcal{A}$ randomly permutes the `edaBits` and then transforms them into circles. $\mathcal{A}$ does the same with the circuits (and corresponding `daBits`), except these are turned into triangles. Now, whether a circle (or triangle) is good or bad depends entirely on whether or not the `edaBit` (or circuit) is consistent or not.

$\mathcal{A}$ sends the set of circles to the challenger of **SimpleGame**, who then throws them randomly in buckets and sends these back to $\mathcal{A}$. The same happens with the set of triangles. In **RealGame**, $\mathcal{A}$ pairs the `edaBits` and the circuits according to the same arrangement as what was given to $\mathcal{A}$ from the **SimpleGame** challenger. These are then given to $\mathcal{B}$. If $\mathcal{B}$ is capable of winning **RealGame**, then $\mathcal{B}$ can be used to win the **SimpleGame**, as the configuration given by $\mathcal{A}$ to $\mathcal{B}$ is indistinguishable from that given by a real challenger of **RealGame**. This is due to the same permutation and due to the function f being used within the **SimpleGame**: f is created specifically to mimic the checking procedure of **RealGame**, so this behaves in an indistinguishable way as well.

If $\mathcal{B}$ wins **RealGame** with non-negligible probability, then $\mathcal{A}$ wins the **SimpleGame** with the same probability. □

A.1 Analysis of the Cut and Choose procedure

We will now prove that this cut-and-choose protocol is sound, as stated in 1.

In order to pass the bucketcheck, the adversarial prover will have to fill every bucket with (ball, ball, triangle) arrangements according to Table 3.

We will now analyze the probability of success of an adversarial prover, i.e. that the prover gets through all three checks described in bucketcheck with at least one inconsistent conversion tuple. Throughout this analysis we will use b to denote the number of bad balls (of the edaBits within checking tuples) and t to denote the number of bad triangles. We assume that $N \geq 2^{\frac{s}{B-1}}$.

First consider the openings taking place during the first two checks.

Opening C balls: In the first check, C of the $NB + C$ balls are opened and checked for consistency. Thus

$$\mathsf{Pr}[C \text{ balls are good}] = \frac{\binom{NB+C-b}{C}}{\binom{NB+C}{C}} \approx \left(1 - \frac{b}{NB+C}\right)^C$$

For $b = (NB + C)\alpha$ (where $1/(NB + C) \leq \alpha \leq 1$), the probability can be written as $(1 - \alpha)^C$. To bound this success probability using the statistical security parameter s, we consider $\alpha \geq \frac{2^{s/B}-1}{2^{s/B}}$ and $C = B$:

$$\mathsf{Pr}[C \text{ balls are good}] \approx (1 - \alpha)^C = (2^{-s/B})^B = 2^{-s}$$

We conclude that if the challenger opens $C = B$ balls, then $\mathcal{A}$ must corrupt less than an α (for $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$) fraction of the balls in order to achieve the respective success probability. We summarize in the following lemma.

Lemma 6. *The probability of $\mathcal{A}$ passing the second check in **SimpleGame** is less than 2^{-s}, if the adversary corrupts more than α fraction of triangles for $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens C triangles.*

Opening C' triangles: The second check if very similar to the first check as the number of balls is the same as the triangles. We therefore arrive at the following statement,

$$\mathsf{Pr}[C' \text{ triangles are good}] = \frac{\binom{NB+C'-t}{C'}}{\binom{NB+C'}{C'}} \approx \left(1 - \frac{t}{NB+C'}\right)^{C'}$$

Using similar argumentation but bounding by a β fraction rather than α and letting $C' = C = B$, we conclude

$$\mathsf{Pr}[C' \text{ triangles are good}] \approx (1 - \beta)^C = (2^{-s/B})^B = 2^{-s}$$

Lemma 7. *The probability of $\mathcal{A}$ passing the second check in **SimpleGame** is less than 2^{-s}, if the adversary corrupts more than β fraction of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens C' triangles.*

The Lemmas 6 & 7 imply that whenever the fraction or bad ball or triangles is large enough, the adversary would already lose during the first two checks. We now analyze the probability of hitting arrangements that pass bucketcheck in such a way that $\mathcal{A}$ wins with respect to small enough fractions of faulty balls and triangles.

Bucketcheck procedure: We here consider the probability of filling a bucket of size B with *bad* balls and triangles as this case may allow the adversary to win with an inconsistent conversion tuple. The challenger has already fixed an arrangement of NB balls into the N buckets. Once this ball arrangement is fixed, it leads to a restriction on the number of favorable arrangements of triangles. As an illustration, consider the following arrangement of 9 balls with $N = 3$ buckets of size 3 and that $\mathcal{A}$ has corrupted $K = 1$ buckets and that this happens to be the first bucket after having been shuffled.

{(●, ○, ●), (●, ○, ○), (○, ○, ○)}

Note that we use different shades of grey for different types of bad balls.

Using Table 3 we see there are two possible favorable combinations of triangles.

{(△, ▲, △), (▲, △, △), (△, △, △)}

{(▲, ▲, ▲), (▲, △, △), (△, △, △)}

This is due to the last entry of Table 3 saying that whenever there are two bad balls, the check may pass using a good triangle ([●, ●, △]) or a bad triangle ([●, ●, ▲]). In this example, let f(●, ●, ▲) $= 1$ (and f(○, ●, ▲) $= 1$) in which case the second arrangement is the favorable arrangement.

As a result of this discussion, the probability of passing $\mathsf{bucketcheck}$ depends on the probability of hitting that specific arrangement of triangles among all possible arrangements of triangles. Thus, the probability of $\mathcal{A}$ passing the last check of $\mathsf{bucketcheck}$ given a specific arrangement of balls L_i is given by

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] \leq 1/\binom{NB}{t}$$

where $t = NB\beta$. Thus,

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] \leq \frac{(NB\beta)! \cdot (NB \cdot (1-\beta))!}{NB!}$$

as we know $0 \leq \beta \leq 1$.

Now, to give an upper bound for $\mathsf{Pr}[\mathcal{A}$ passes $\mathsf{bucketcheck}]$ we provide an upper bound of the probability for different ranges of α and β where the total probability is given by

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] = \sum_i \mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] \cdot \mathsf{Pr}[L_i]$$

where L_i is a given arrangement. If we can then argue for all possible $\frac{1}{NB} \leq \alpha \leq \frac{2^{s/B}-1}{2^{s/B}}$, the probability for $\mathsf{Pr}[\mathcal{A}$ passes $\mathsf{bucketcheck}|L_i]$ (for some configuration L_i), can be bounded by 2^{-s}, then

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] \leq \sum_i 2^{-s} \cdot \mathsf{Pr}[L_i]$$

We now try to bound $\mathsf{Pr}[\mathcal{A}$ passes $\mathsf{bucketcheck}|L_i]$. To this end, we will consider three different ranges from which t might come from, as defined by the monotonicity of the binomial coefficient $\binom{NB}{t}$.

Case I. Let $B \leq t \leq (NB - B)$. Now

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] \leq 1/\binom{NB}{t}$$

This probability is maximal at $t = B$ or $t = NB - B$ as given by

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] = \frac{B! \cdot (NB - B)!}{NB!}$$

$$= \frac{B}{NB} \cdot \frac{B-1}{NB-1} \cdots \frac{1}{NB-(B-1)}$$

Now, given that $N \geq 2^{s/B}$, we arrive at

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck} | L_i] \leq \left(\frac{1}{s^{s/B}}\right)^B = 2^{-s}$$

Lastly we note that

$$\mathsf{Pr}[L_i] = \frac{(NB-b)!}{NB!}$$

where b describes the number of bad balls (and in turn $NB - b$ is the number of good balls). Combining these two last equations, we conclude

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] \leq \frac{NB!}{(NB-b)!} \cdot 2^{-s} \cdot \frac{(NB-b)!}{NB!}$$

when $t \in [B, NB - B]$.

Case II. Let $t > NB - B$. Whenever t is greater than $NB - B$, $\mathcal{A}$ will not be able to pass the initial phase where $C = B$ triangles are opened.

Case III. Due to this type of game being very difficult to analyse generally, we instead consider it for the specific bucket sizes in our theorem statement. We will begin by looking at a bucket size of 3 and then analyse $t = 0, t = 1$ and $t = 2$.

Bucket size 3: For a bucket of size 3 and $t = 0$ the analysis has already been done in [EGK+20], who show that the success probability is $< 2^{-s}$.

For $t = 1$, however, the adversary could now also hope to place a bad triangle in the top spot of a bucket. This does not change the number of cases though, as the case of having a good ball and a bad triangle in the top spot is equivalent to having a good ball and a bad triangle in any other spot of the bucket. Furthermore, increasing the amount of triangles does not increase the probability of $\mathcal{A}$ hitting a favorable permutation (as already argued in **Case I**), we conclude that $t = 1$ remains similar to [EGK+20].

Lastly, we consider $t = 2$. Now the adversary has to compensate for an extra bad triangle, compared to the previous case. In this case we encounter a specific arrangement that could be an issue regarding our way of analysing these cases. We will argue, however, that this is not a problem.

The following arrangement

$$\{(●, ●, ●), (●, ●, ●), (○, ●, ○), (○, ○, ○)\}$$

could lead to an adversary filling up more buckets than intended. Let a predefined K such that $1 \leq K \leq N-1$ exist such that $\mathcal{A}$ wants to pass the test with K bad conversion tuples. Now, we could define $b = KB + 1$ bad balls for $K = 2$ and then with $t = 2$ arrive at the above arrangement. This arrangement however, allows $\mathcal{A}$ to actually cheat in 3 buckets rather than 2, since if $\mathcal{A}$ hits the following arrangement of triangles

$$\{(△, △, △), (△, △, △), (▲, △, ▲), (△, △, △)\}$$

the bucket check could pass with three conversion tuples instead. To remedy this situation, observe that $\mathcal{A}$ does not simply win by filling buckets with bad balls and triangles, but more specifically these buckets must also contain a corrupt conversion tuple. Therefore, if $\mathcal{A}$ only created $K = 2$ inconsistent conversion tuples (and let's assume these are within the first two buckets), then the third bucket will fail, regardless of the triangles hitting the correct arrangement to satisfy three bad elements and in turn three bad buckets. As such, this would not be a *favorable* arrangement.

When $B = 3$ and $t = 2$, there are a total of 6 favorable arrangements for $\mathcal{A}$ when K is fixed. For example, for $N = 4$, $B = 3$, $K = 2$, $t = 2$, these are the six possible configurations that are favorable for $\mathcal{A}$.

{(◍, ◍, ◍), (●, ●, ◍), (○, ○, ○), (○, ○, ○)}

{(◍, ◍, ◍), (◍, ●, ◍), (○, ◍, ○), (○, ○, ○)}

{(◍, ◍, ◍), (◍, ●, ○), (○, ○, ○), (○, ○, ○)}

{(◍, ◍, ◍), (◍, ○, ○), (○, ○, ○), (○, ○, ○)}

{(◍, ◍, ◍), (◍, ◍, ●), (○, ◍, ○), (○, ○, ○)}

{(◍, ◍, ◍), (◍, ◍, ◍), (○, ◍, ◍), (○, ○, ○)}

Note that the darker balls ● are such that f(◍, ●, ▲) $= 1$ in all the listed configurations, forcing $\mathcal{A}$ to specifically hit the ▲ in those spots (i.e. forcing that only a single permutation of the triangles will be favorable).

For all of the above cases, the success probability of $\mathcal{A}$ in the bucket check can be expressed as

$$\Pr[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] \leq \binom{N}{K}\binom{KB}{g_1+b_1}\binom{(N-K)B}{b_2}\frac{1}{\binom{NB}{KB-g_1+b_2}}\frac{1}{\binom{NB}{t}} \tag{2}$$

where g_1 is the total number of good balls in the K buckets containing bad conversion tuples, b_1 is the total number bad balls of a different kind from ◍ (represented as ●) and b_2 is the total number of bad balls having been placed in the remaining $N-K$ buckets containing good conversion tuples. Now, for each possible configuration (when varying g_1, b_1, b_2 and K but keeping $t = 2$ static), the probability of $\mathcal{A}$ winning is maximum at $K = 1$ or $K = N-1$. Considering all possible favorable configurations, the second that we list (with $g_1 = 0, b_1 = 1, b_2 = 1$) has the highest probability of success with $K = N-1$.

$$\begin{aligned}
&\binom{N}{N-1}\binom{3(N-1)}{1}\binom{3(N-(N-1))}{1}\frac{1}{\binom{3N}{3(N-1)+1}}\frac{1}{\binom{3N}{2}} \\
&= N\cdot(3N-3)\cdot 3\frac{1}{\binom{3N}{3N-2}}\frac{1}{\binom{3N}{2}} \\
&= \frac{3N-3}{3N-1}\cdot\frac{2}{3N}\cdot\frac{2}{3N-1} \\
&\leq \frac{3N-3}{3N-1}\cdot 2^{-s/2}\cdot 2^{-s/2} \leq 2^{-s}, \text{ given } N \geq 2^{s/2}
\end{aligned}$$

Bucket size 4: We've already considered the cases of $t = 0$, $t = 1$ and $t = 2$, so now we'll consider $t = 3$.

For the case when $B = 4$ and $t = 3$ there are a total of 10 favorable configurations for $\mathcal{A}$ when K is fixed. For example, for $N = 4$ and $K = 2$, there are the 10 cases:

{(◍, ◍, ◍, ◍), (◍, ○, ○, ○), (○, ○, ○, ○), (○, ○, ○, ○)}

{(◍, ◍, ◍, ◍), (◍, ◍, ○, ○), (◍, ○, ○, ○), (○, ○, ○, ○)}

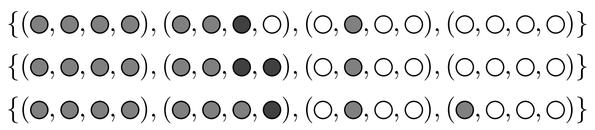
{(◍, ◍, ◍, ◍), (◍, ◍, ◍, ○), (◍, ○, ○, ○), (○, ◍, ○, ○)}

{(◍, ◍, ◍, ◍), (◍, ◍, ◍, ◍), (◍, ○, ○, ○), (○, ◍, ◍, ○)}

{(◍, ◍, ◍, ◍), (◍, ●, ○, ○), (○, ○, ○, ○), (○, ○, ○, ○)}

{(◍, ◍, ◍, ◍), (◍, ●, ●, ○), (○, ○, ○, ○), (○, ○, ○, ○)}

{(◍, ◍, ◍, ◍), (◍, ●, ●, ●), (○, ○, ○, ○), (○, ○, ○, ○)}

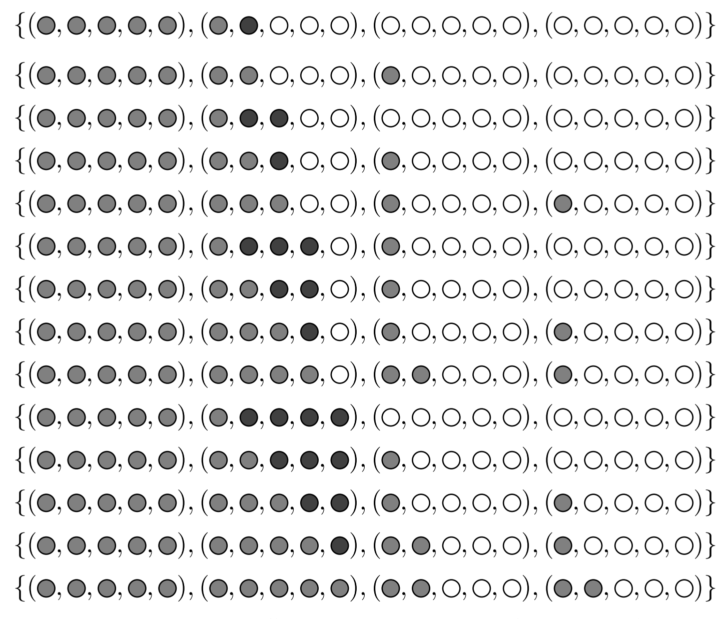


Using eq. 2 we can compute the probabilities of success for all these cases at $K = 1$ and $K = N - 1$ in order to find the best possible scenario for $\mathcal{A}$. By doing so, we conclude that the 9'th case has the highest probability of success at $K = N - 1$. Considering this case, the probability is computed as,

$$\begin{aligned}\Pr[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] &\le \binom{N}{N-1}\binom{(N-1)\cdot 4}{2} \cdot \binom{4}{1} \frac{1}{\binom{4N}{4(N-1)+1}} \frac{1}{\binom{4N}{3}} \\ &= N \cdot \binom{4N-4}{2} \cdot 4 \cdot \frac{(4N-3)! \cdot 3!)}{(4N)!} \cdot \frac{3! \cdot (4N-3)!}{(4N)!} \\ &= 8(N-1) \cdot (4N-5) \cdot \frac{3}{4N} \cdot \frac{3}{4N} \cdot \frac{2}{4N-1} \cdot \frac{2}{4N-1} \cdot \frac{1}{4N-2} \cdot \frac{1}{4N-2} \\ &\le 8(N-1) \cdot (4N-5) \cdot (2^{-s/3})^6, \text{ given } N \ge 2^{s/3}\end{aligned}$$

Bucket size 5: As per the last bucket size, the analysis from the previous cases carries over for $t = 0, 1, 2$ and 3. In this case, we will consider $t = 4$, as this is the only remaining for $t < B$.

For the case where $B = 5$ and $t = 4$, there are 15 favorable arrangements for $\mathcal{A}$ when K is a fixed integer. For instance, for $N = 4$ and $K = 2$, these are the cases.



Again, by using eq. 2, we compute the different probabilities of success of all 15 cases in similar fashion to our analysis of $B = 3$ and $B = 4$ and conclude that the 13'th case has the highest probability of winning at $K = N - 1$. We now describe this probability.

$$\Pr[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] \le \binom{N}{N-1}\binom{(N-1)\cdot 5}{2} \cdot \binom{5}{2} \frac{1}{\binom{5N}{5(N-1)+2}} \frac{1}{\binom{5N}{4}}$$

$$= N \cdot \binom{5N-5}{2} \cdot \binom{5}{2} \cdot \frac{(5N-3)! \cdot 3!}{(5N)!} \cdot \frac{4! \cdot (5N-4)!}{(5N)!} \le 2^{-s}, \text{ given } N \ge 2^{s/4}$$

We summarize the analysis as follows.

Lemma 8. *The probability of $\mathcal{A}$ passing* bucketcheck *in **SimpleGame** is less than 2^{-s} given $N \geq 2^{s/B-1}$ and the challenger opens $C = B$ balls and $C' = B$ triangles during the first two checks of **SimpleGame** for $B \in \{3, 4, 5\}$ given s such that $\frac{s}{B-1} > B$.*

Proof. This lemma follows from a case-by-case analysis of the bucketcheck procedure, in combination with Lemmas 6 & 7. □

Combining Lemma 5 and Lemma 8 completes the proof of Theorem 1.

A.2 Proof of Security of the Protocol Π_{Conv} (Theorem 2)

Proof. We first consider a malicious prover and then afterwards we consider the case of a malicious verifier. In both cases we construct a simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{Conv}}$ that runs the adversary $\mathcal{A}$ as a subroutine. We implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and $\mathcal{Z}$.
Malicious Prover. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{P})$ to the ideal functionality $\mathcal{F}_{\mathsf{Conv}}$. $\mathcal{S}$ creates a copy of the verifier $\mathcal{V}$, and runs this verifier according to the protocol Π_{Conv}, while letting the prover $\mathcal{P}^*$ behave as instructed by the adversary $\mathcal{A}$.

1. In the setup-phase, $\mathcal{P}^*$ sends `edaBits` $\{(r_0^j, \ldots, r_{m-1}^j, r^j)\}_{j\in[NB+C]}$, `daBits` $\{(b^j, b^{',j})\}_{j\in[NB+s]}$ and triples $\{(x^j, y^j, z^j)\}_{j\in[NBm+Cm]}$. $\mathcal{S}$ records and forwards all of these to $\mathcal{F}_{\mathsf{Conv}}$.
2. $\mathcal{S}$ runs $\mathcal{F}_{\mathsf{Dabit}}$ with the input provided by $\mathcal{P}^*$. If $\mathcal{F}_{\mathsf{Dabit}}$ returns abort, then send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate. Otherwise continue.
3. $\mathcal{S}$ randomly sample permutations $\pi_1 \in S_{NB+C}, \pi_2 \in S_{NB+s}$ and $\pi_3 \in S_{NBm+Cm}$ and send these to $\mathcal{P}^*$ before shuffling the values provided by $\mathcal{P}^*$.
4. The simulator emulates the **Cut**-phase by calling Open on the last C `edaBits` and triples and ensuring the consistency of each. If any check fail, send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate.
5. For each bucket during the **Choose**-phase, $\mathcal{S}$ emulate bitADDcarry by running like an honest verifier and convertBit2A by calling this using $\mathcal{F}_{\mathsf{Conv}}$.
6. $\mathcal{S}$ runs the rest of the protocol as an honest verifier. If the honest verifier outputs abort, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate. If the honest verifier outputs (success), then $\mathcal{S}$ sends (VerifyConv, $N, \{[c_0^{(j)}]_2, \ldots, [c_{m-1}^{(j)}]_2, [c^{(j)}]_M\}_{j\in[N]}$) to $\mathcal{F}_{\mathsf{Conv}}$ (essentially forwarding the call made originally by $\mathcal{P}^*$).

The messages that $\mathcal{P}^*$ receives from $\mathcal{S}$ have the same distribution as in the real protocols. Whenever the verifier simulated by $\mathcal{S}$ outputs abort (as in the protocol), then the verifier in the ideal setting outputs abort as well (since $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Conv}}$). The only case where $\mathcal{P}^*$ may distinguish between the ideal and real, is if the simulated verifier run by $\mathcal{S}$ outputs (success) but at least one conversion tuple is inconsistent, in which case $\mathcal{F}_{\mathsf{Conv}}$ will abort. But if *at least* one conversion is inconsistent, then by Theorem 1 the probability with which $\mathcal{P}^*$ avoids being caught in Step 6 of Π_{Conv} is at most 2^{-s}.

Malicious Verifier. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}_{\mathsf{Conv}}$. It also creates copies of the prover $\mathcal{P}$ and verifier $\mathcal{V}^*$, and runs the prover according to the protocol Π_{Conv}, while letting the verifier behave as instructed by the environment $\mathcal{Z}$. If $\mathcal{S}$ receives abort from $\mathcal{F}_{\mathsf{Conv}}$, then it simply outputs abort and terminate. Otherwise $\mathcal{S}$ interacts with the verifier as follows:

1. $\mathcal{S}$ samples random values corresponding to the `edaBits` $\{(r_0^j, \ldots, r_{m-1}^j, r^j)\}_{j\in[NB+C]}$, `daBits` $\{(b^j, b^j)\}_{j\in[NB+s]}$ and triples $\{(x^j, y^j, z^j)\}_{j\in[NBm+Cm]}$ and commits to these by calling $(\mathsf{Input}, \cdot, \cdot)$ using $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ in the appropriate domain.
2. If $\mathcal{V}^*$ at any outputs abort, then send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate.

3. $\mathcal{S}$ sends $(\mathsf{VerifyDabit}, NB+s, \{([b^j]_2, [b^{',j}]_M)\}_{j\in[NB+s]})$. If this call returns abort, then output abort and terminate. Otherwise continue.

4. On receiving π_1, π_2, π_3, $\mathcal{S}$ locally shuffles the sampled values.

5. During the **Cut**-phase, $\mathcal{S}$ opens the last C `edaBits`, triples and `daBits` honestly, as it knows the underlying values.

6. During the remainder of the protocol, $\mathcal{S}$ runs like an honest prover.

7. Lastly, $\mathcal{S}$ sends
$$(\mathsf{VerifyConv}, N, \{[c_0^{(j)}]_2, \ldots, [c_{m-1}^{(j)}]_2, [c^{(j)}]_M\}_{j\in[N]})$$
to $\mathcal{F}_{\mathsf{Conv}}$ (again, forwarding the initial call) and outputs whatever $\mathcal{V}$ outputs.

In both the ideal and real execution all values sent from the honest prover (or simulator) to the verifier, are hidden by $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$. This ensures indistinguishability between the two transcripts. Specifically, the randomly sampled `edaBits`, triples and `daBits` are indistinguishable from ones sampled by the prover during a real execution, due to $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$. Any calls to $\mathsf{VerifyDabit}$ will fail in the real only if the same call would fail in the ideal world, due to the usage of $\mathcal{F}_{\mathsf{FDabit}}$ in both. Same goes for $\mathsf{CheckZero}$. Lastly, $\mathsf{bitADDcarry}$ only involves the prover sending values to the verifier, allowing the verifier to reach the same value. An honest prover uses correct circuits and therefore no information is leaked. Thus, the view of $\mathcal{V}^*$ simulated by $\mathcal{S}$ is distributed identically to its view in the real protocol execution.

□

B Proof of Conversion Protocol for Faulty Dabits

First we present the ideal functionality $\mathcal{F}_{\mathsf{FDabit}}$ (Figure 20) and then prove Lemma 1. We then argue that we can use these faulty `daBits` within the protocol Π_{Conv}, by looking at what potential errors might occur when using these, creating a revised **RealGame** called **RealGame**$_{\texttt{faulty}}$ and then providing a proof sketch reducing the **RealGame**$_{\texttt{faulty}}$ to the **SimpleGame** (Figure 18) of the original proof in Lemma 5.

Functionality $\mathcal{F}_{\mathsf{FDabit}}$

This functionality extends $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ with the extra function $\mathsf{VerifyDabit}$ that takes a set of IDs $\{(\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]}$ of (potentially faulty) `daBits` $\{(b^{\mathsf{id}^{0,j}}, b^{\mathsf{id}^{1,j}})\}_{j\in[N]}$ and verifies that $b^{\mathsf{id}^{0,j}} \equiv b^{\mathsf{id}^{1,j}} \bmod 2$ where $b^{\mathsf{id}^{0,j}} \in \mathbb{Z}_2$ an $b^{\mathsf{id}^{1,j}} \in \mathbb{Z}_M$ for all $j \in [N]$. It is assumed that the id's of the potential `daBits` have been Input prior to execution of this method. $\mathcal{F}_{\mathsf{FDabit}}$ communicates with two parties $\mathcal{P}, \mathcal{V}$.

Verify: On input $(\mathsf{VerifyDabit}, N, \{(\mathsf{id}^{0,j}, \mathsf{id}^{1,j})\}_{j\in[N]})$ by $\mathcal{P}$ and $\mathcal{V}$ (where $(\mathsf{id}^{0,j}, b^{\mathsf{id}^{0,j}}), (\mathsf{id}^{1,j}, b^{\mathsf{id}^{1,j}}) \in S$ for $j \in [N]$).

- If $b^{\mathsf{id}_{0,j}} \equiv b^{\mathsf{id}_{1,j}} \bmod 2$ for all $j \in [N]$, then output $(\mathsf{success})$ to $\mathcal{V}$, otherwise output abort.

Figure 20: Ideal functionality modeling the verification of faulty `daBits`.

The proof of Lemma 1 uses the following simple proposition:

Proposition 1. *For all $b_1, \ldots, b_N \in \mathbb{Z}_p$ it holds that*

$$\Pr_{e_1,\ldots,e_N \leftarrow \{0,1\}}[|(\sum_{i=1}^{N} e_i b_i \bmod p)| < 1/2 \cdot \max\{|b_i|\}] \leq 1/2.$$

A proof for this can e.g. be found in [BL17, Lemma 2.3].

Proof of Lemma 1 (Sketch). We first consider a malicious prover and then afterwards a malicious verifier. In both cases we argue intuitively for the existence of a correct simulator that, given access to $\mathcal{F}_{\mathsf{FDabit}}$ and emulating $\mathcal{F}^{2,M}_{\mathsf{ComZK}}$ creates a correct distribution of transcripts.

Malicious Prover. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{P})$ to the ideal functionality $\mathcal{F}_{\mathsf{FDabit}}$. It also creates copies of the prover $\mathcal{P}^*$ and verifier $\mathcal{V}$, and runs the verifier according to the protocol Π_{FDabit}, while letting the prover behave as instructed by the environment $\mathcal{Z}$.

1. $\mathcal{S}$ gives the provided set of id's to $\mathcal{F}_{\mathsf{FDabit}}$, given what $\mathcal{P}^*$ inputs into $\mathcal{F}^{2,M}_{\mathsf{ComZK}}$.
2. For each of the s iterations, $\mathcal{S}$ follows the protocol honestly.
3. If the protocol succeeds, then it runs $\mathsf{VerifyDabit}$ in $\mathcal{F}_{\mathsf{FDabit}}$, otherwise it sends abort.

As the $\mathcal{S}$ honestly provides N random bits in each iteration and nothing else is sent by the honest verifier, the view of $\mathcal{P}^*$ is distributed equally to the real world. $\mathcal{Z}$ can only distinguish in the case of the passing the check in Π_{FDabit} with a `daBit` $([b]_2, [b']_M)$ such that $b \not\equiv b' \bmod 2$, as this would not pass in the ideal world.

Consider the sum $r' = \sum_{i=1}^{N} e_i b_i$. Then since $0 \leq r' + \sum_{i=1}^{\gamma} c_i 2^{i-1} < 2^{\gamma}$ for $c_i \in \{0,1\}$ we know that $-2^{\gamma} \leq r' < 2^{\gamma}$ must hold (given then lower bound on p no wraparound can occur). As we compute r' as in Proposition 1 in each of the s rounds, it must hold that $\max_{i\in[N]}\{|b_i|\} \leq 2^{\gamma+1}$ except with probability 2^{-s}. But then, by definition, $\tau = \sum_{i=1}^{N} e_i b_i + \sum_{i=1}^{\gamma} c_i 2^{i-1}$ holds over the integers, and in particular it holds that $r = \tau = c_0 + \sum_{i=1}^{N} e_i b_i \bmod 2$. Now, except with probability 2^{-s}, the claim on the parity of the `daBits` follows by a standard argument.

Malicious Verifier. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}_{\mathsf{FDabit}}$. It also creates copies of the prover $\mathcal{P}$ and verifier $\mathcal{V}^*$, and runs the prover according to the protocol Π_{FDabit}, while letting the verifier behave as instructed by the environment $\mathcal{Z}$.

1. $\mathcal{S}$ sends $(\mathsf{VerifyDabit}, N, \{[b^j]_2, [b^{',j}]_M\})$ to $\mathcal{F}_{\mathsf{FDabit}}$. If $\mathcal{F}_{\mathsf{FDabit}}$ returns abort, then $\mathcal{S}$ outputs abort to $\mathcal{V}^*$ and terminate. Otherwise continue.
2. Since $\mathcal{F}_{\mathsf{FDabit}}$ did not abort, we know that it holds for each $j \in [N]$ that $b^j \equiv b^{',j} \bmod 2$. The simulator executes the remainder of the protocol Π_{FDabit} as follows:
   - For r it opens a uniformly random bit.
   - Then it reveals a uniformly random $\tau \in [0, 2^{\gamma} - 1]$ such that $\tau = r \bmod 2$.

The view of $\mathcal{V}^*$ simulated by $\mathcal{S}$ is distributed equally to its view in the real world. We consider what the corrupted verifier $\mathcal{V}^*$ sees during an execution of Π_{FDabit} and compare this to the simulation. In an honest protocol instance, c_0 is uniformly randomly chosen and therefore the revealed value r is also uniformly random. This is also true for the simulation. Now, observe that $r' \in [0, N]$ in the protocol. The protocol adds a uniformly random value from $[0, 2^{\gamma} - 1]$ to r', which hides all information about r' by so-called noise drowning but makes the lowest bit agree with r. This is then indistinguishable from a value as sampled in the simulation, as $\tau \geq 2^{\gamma}$ can only occur in the real protocol with probability 2^{-s}. □

We now in more detail consider what potential errors can be caused by using a faulty multiplication triple during the $\mathsf{bitADDcarry}$ procedure. The purpose of this discussion is to distinguish between the potential errors caused by a faulty `daBit` and those caused by a faulty multiplication triple.

Faulty Multiplication Triples: A single faulty multiplication triple will allow for the change of the output of $\mathsf{bitADDcarry}$ (as described in Figure 7 and the appendix in Section E.1) in such a way that it results in a single bit being flipped (and in turn cause several bits after this to be flipped due to the usage of a ripple-carry adder). As such, the output will still be guaranteed to be bits, but if these were to be summed up (as does happen in the final check of Π_{Conv}), the error essentially causes the adding or subtraction of an arbitrary value $\mathtt{err}_{\mathtt{triples}} < 2^m$ (although the result will always be positive, as the output of the ripple-carry adder is treated as unsigned).

If we consider this in the context of the consistency check, a faulty multiplication triple means that either some of the authenticated bits $[e_i]_2$ for $i = 0, \dots, m-1$ and/or the carry bit $[e_m]_2$ is incorrect. This error can then be used to force equality of the check $e' \overset{?}{=} \sum_{i=0}^{m-1} 2^i \cdot e_i$ (for $e' = [r+s]_M - 2^m \cdot [e_m]_M$) in the case of an inconsistent conversion tuple, by masking the error $\mathtt{err}_{\mathtt{triples}} < 2^m$, thus allowing an adversarial prover to pass the check without a consistent conversion tuple. Thus, a tampered circuit (or rather inconsistent multiplication triple) only allows for the flipping of output bits, meaning that we are guaranteed that any output from the $\mathsf{bitADDcarry}$ evaluation will be bits, although they may lead to arbitrary error $0 \leq \mathtt{err}_{\mathtt{triples}} < 2^m$.

Faulty `daBits`: We now look at what can happen when faulty `daBits` are introduced. The only function that utilizes `daBits` within Π_{Conv} is the $\mathsf{convertBit2A}$ procedure as defined in Figure 6.

In $\mathsf{convertBit2A}$, a part of the conversion of a bit from an authentication in $\mathbb{Z}_2$ to one in $\mathbb{Z}_M$ requires opening an addition of two supposed bits (step 1) which will always yield a bit value c. Therefore, only the arithmetic part of the `daBit` $([b]_2, [b']_M)$ may be incorrect. Let $[b']_M$ be an authentication of a different value from $[b]_2$. This gives two cases.

1. $[b']_M$ is a bit ($[b]_2$ flipped),
2. $[b']_M$ is not a bit.

If $b' \in \{0,1\}$ for $b' \neq b$, then the output $[x']_M$ will be the input bit $[x]_2$ flipped. We show the computation for one example, but note that this will happen with both combinations $((b=0, b'=1)$ or $(b=1, b'=0))$.

Let $x = 0$ and $(b = 0, b' = 1)$.

1. $c = b \oplus x = 0 \oplus 0 = 0$
2. $x' = c + r' - 2 \cdot c \cdot r' = 0 + 1 - 2 \cdot 0 \cdot 1 = 1$

We now argue why this particular type of error can break the security proof of Π_{Conv} in Section 3.3. Consider the case where the carry bit of $\mathsf{bitADDcarry}$ is tampered such that it would have been $x = 0$, but was flipped to $x_{\mathtt{err}} = 1$. When $x_{\mathtt{err}}$ is then input to $\mathsf{convertBit2A}$, it is flipped back to $x' = x = 0$ if a faulty `daBit` $([b]_2, [b']_M)$ such that $b' \in \{0,1\}$ but $b' \neq b$ is used. This behavior proofs to be detrimental to the analysis in Section 3.3: now, a conversion tuple check might pass with either a good triple, or an inconsistent triple in the case of this triple being negated by the call to $\mathsf{convertBit2A}$. This provides an adversarial prover more options in winning **SimpleGame** and as much must be disallowed.

We now consider what happens in $\mathsf{bitADDcarry}$ when using a `daBit` $([b]_2, [b']_M)$ where $b' \notin \{0,1\}$. Define $\delta = (-1)^b(b' - b)$. Then the output of $\mathsf{convertBit2A}$ using x as input is $x_{\mathtt{err}} = x + (-1)^x \cdot \delta$.

This $x_{\mathtt{err}}$ (authenticated in $\mathbb{Z}_M$) is then used in the computation $[c']_M = [r+s]_M - 2^m \cdot x_{\mathtt{err}}$ which is meant to remove the potential carry bit from the result of $[r+s]_M$ (in the case of a correct `daBit`), such that we can check if $c' \overset{?}{=} \sum_{i=0}^{m-1} 2^i \cdot c_i$ (where c_i, $i = 0, \dots, m-1$ are the bits from the sub-protocol $\mathsf{bitADDcarry}$). Now, if $x_{\mathtt{err}}$ isn't a bit, this causes c' to be either larger (or smaller) than $\sum_{i=0}^{m-1} 2^i \cdot c_i$ due to $2^m \cdot x_{\mathtt{err}}$ being much larger than anything which can be represented in m bits, given that $x_{\mathtt{err}} \notin \{0,1\}$. To this end, we conclude that a faulty `daBit` such that $x_{\mathtt{err}} >\notin \{0,1\}$ will allow an adversary to pass Π_{Conv} using an inconsistent conversion tuple $([r_0]_2, \dots, [r_{m-1}]_2, [r]_M)$ with an error of $\mathtt{err}_{\mathtt{daBit}} = r - \sum_{i=0}^{m-1} r_i \geq 2^m$.

We sum up this result in the following lemma.

Lemma 9. *For any inconsistent conversion tuple* $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)$ *such that* $(r - \sum_{i=0}^{m-1} r_i) = \texttt{err} \notin \{0,1\}$ *designed to pass* Π_{Conv} *using faulty multiplication triples, it must hold that* $\texttt{err} < 2^m$*, while if the conversion tuple was designed to pass using faulty dabits,* $\texttt{err} \geq 2^m$*.*

B.1 Security Analysis of the Faulty Dabit Approach

To accommodate for the `daBits` used within the conversion verification, we now modify **RealGame** (Figure 17) . The purpose is to treat `daBits` (which may now be inconsistent) separately, but still have them be merged with the faulty addition circuits in order to be able to reduce the security of this new **RealGame**$_{\texttt{faulty}}$ (Figure 21) to that of our old **SimpleGame** (Figure 18). By having `daBits` and circuits be combined, we can treat their combination as a single entity such that we can directly describe **RealGame**$_{\texttt{faulty}}$ in terms of the **SimpleGame** syntax from the original proof. We formally define the revised **RealGame**$_{\texttt{faulty}}$ in Figure 21. The major difference is found in step 4, but in addition to this last check being altered, the challenger must also open C'' of the `daBits`.

For brevity, we only list the ways in which this new **RealGame**$_{\texttt{faulty}}$ differs from the **RealGame** defined in Figure 17.

The RealGame$_{\texttt{faulty}}$ with faulty `daBits`

1. $\mathcal{A}$ prepares must now prepare $NB + C$ `daBits` $\{([b^j]_2, [b^{',j}]_M)\}_{j \in [NB+C]}$.
2. The challenger opens C `edaBits`, circuits and `daBits` now. If any of the `edaBits`, `daBits` or circuits are inconsistent, abort.
3. The challenger pairs up the remaining NB circuits with the remaining NB `daBits` according to their permutations.
4. (Modified step 6 of original **RealGame**) Within each of the buckets, for every pair of `edaBits` $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)$ (where this is the first of the bucket) and $([s_0]_2, \ldots, [s_{m-1}]_2, [s]_M)$ take the next circuit C^* and `daBit` (b_2, b_M) and compute $(c_0, \ldots, c'_m) \leftarrow C^*(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1}, (b_2, b_M))$. Define $\delta = (-1)^{b_2} \cdot (b_M - b_2)$ and then let $c'_m = c_m + (-1)^{c_m} \cdot \delta$ where c_m is the correct result. The circuit C^* works as a tamper-resilient binary addition circuit, but a potential error `err` is added to the carry bit c_m if the `daBit` (b_2, b_M) is inconsistent. If the `daBit` is consistent then $c'_m = c_m$. Then compute $c = \sum_{i=0}^{m-1} 2^i \cdot c_i$ and check $c \overset{?}{=} r + s - 2^m \cdot c'_m$.

Figure 21: The revised **RealGame**$_{\texttt{faulty}}$ which handles (potentially faulty) `daBits` separate from the circuits

Note that the term $c'_m = c_m + (-1)^{c_m} \cdot \delta$ corresponds to the evaluation of

$$c'_m \leftarrow c + [b]_M - 2 \cdot c \cdot [b]_M$$

when $[b]_M$ is not an authentication of a bit, which is used during convertBit2A.

We now make a very similar argument to that of Lemma 5, but for the new **RealGame**$_{\texttt{faulty}}$.

Lemma 10. *Security against all adversaries of the* ***SimpleGame*** *implies security against all adversaries of the* **RealGame**$_{\texttt{faulty}}$*.*

Proof. The only difference between this reduction and the original (Lemma 5), is that $\mathcal{A}$ now pairs up circuits and `daBits` and converts these into triangles, rather than only the circuits.

□

As the inconsistency $\mathtt{err}_{\mathtt{daBit}}$ in conversion tuples that may pass Π_{Conv} using a faulty $\mathtt{daBit}$ differs enough from the inconsistency $\mathtt{err}_{\mathtt{triples}}$ allowed by faulty multiplication triples (specifically $\mathtt{err}_{\mathtt{triples}} < \mathtt{err}_{\mathtt{daBit}}$), we conclude that we can treat them as one entity comparative to the tamper-resilient circuits of the original **RealGame**, simply allowing for a wider range of errors. This means the above proof sketch is a sufficient reduction from the $\textbf{RealGame}_{\mathtt{faulty}}$ to the **SimpleGame**, allowing us to re-use our proof from Lemma 5.

C Proof of Correct Truncation

Theorem 3. *The protocol* $\Pi_{\mathsf{VerifyTrunc}}$ *(Figure 12) UC-realizes* $\mathcal{F}_{\mathsf{VerifyTrunc}}$ *(Figure 9) in the* $\mathcal{F}_{\mathsf{CheckLength}}$*-hybrid model.*

Before writing the proof, we make the following observations. First, if correct information is provided by $\mathcal{P}$, then the protocol completes. Intuitively, if the prover provides a correct $[a']_M = [a \mod 2^m]_M$ and $[a_{tr}]_M$, then when both of these are subtracted from $[a]_M$, then it will be equal to 0 as required by CheckZero.

CheckLength **on** $([a']_M, m)$**:** This ensures that $[a']_M$ can be represented by m bits.

CheckLength **on** $([a_{tr}]_M, l-m)$**:** This ensures that $[a_{tr}]_M$ can be represented by $l-m$ bits.

CheckZero$([a]_M - (2^m \cdot [a_{tr}]_M + [a']_M))$**:** This check ensures correctness of the two values $[a']_M$ and $[a_{tr}]_M$. As we know that they are both of correct length (m and $l-m$ respectively), $2^m \cdot a_{tr} + a'$ exactly represents all values in $[0, 2^l - 1]$. Therefore, the truncation must be correct.

We now proceed with the proof.

Proof. We consider a malicious prover and a malicious verifier separately. In both cases we will construct a simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{VerifyTrunc}}$ that will emulate $\mathcal{F}_{\mathsf{CheckLength}}$. We implicitly assume that $\mathcal{S}$ passes all communication between the adversary (either $\mathcal{P}^*$ or $\mathcal{V}^*$ dependent on the case) and the environment $\mathcal{Z}$.

Malicious Prover. $\mathcal{S}$ sends (corrupted, $\mathcal{P}$) to the ideal functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$. It also creates copies of the prover $\mathcal{P}^*$ and verifier $\mathcal{V}$, and runs the verifier according to the protocol $\Pi_{\mathsf{VerifyTrunc}}$, while letting the prover behave as instructed by the environment $\mathcal{Z}$.

1. $\mathcal{S}$ forwards Input on $[a']_M$.
2. $\mathcal{S}$ forwards any calls to $\mathcal{F}_{\mathsf{CheckLength}}$. If any calls to $\mathcal{F}_{\mathsf{CheckLength}}$ returns $\perp$, then $\mathcal{S}$ outputs $\perp$ to $\mathcal{F}_{\mathsf{VerifyTrunc}}$ and abort.
3. For the remainder of the protocol, $\mathcal{S}$ acts like an honest verifier.
4. Lastly, $\mathcal{S}$ forwards the call (VerifyTrunc, $\cdot, \cdot$).

The only avenue for $\mathcal{P}^*$ to distinguish the ideal from the real world is the case of passing the verification check with an incorrect truncation. As argued above, this can never happen. This completes the proof for the case of a malicious prover.

Malicious Verifier. $\mathcal{S}$ sends (corrupted, $\mathcal{V}$) to the ideal functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$. It also creates copies of the prover $\mathcal{P}$ and verifier $\mathcal{V}^*$, and runs the prover according to the protocol $\Pi_{\mathsf{VerifyTrunc}}$, while letting the verifier behave as instructed by the environment $\mathcal{Z}$. If $\mathcal{S}$ receives $\perp$ from $\mathcal{F}_{\mathsf{Conv}}$, then it simply abort. Otherwise $\mathcal{S}$ interacts with the verifier as follows:

1. $\mathcal{S}$ forwards the call (VerifyTrunc, $N, \{m^j, [a^j]_M, [a^j_{tr}]_M\}_{j \in [N]}$). If $\mathcal{F}_{\mathsf{VerifyTrunc}}$ returns $\perp$, output $\perp$ to $\mathcal{V}^*$ and abort.
2. For each $j \in [N]$ $\mathcal{S}$ commits to a random value $[a']_M$ using Input of $\mathcal{F}_{\mathsf{CheckLength}}$. We assume that simulated commitments to $a^j, a^{',j}$ already exist in $\mathcal{F}_{\mathsf{CheckLength}}$.

3. For each iteration $j \in [N]$, let l be the size of a^j and m be the size of $a^{',j}$. $\mathcal{S}$ runs $(\mathsf{CheckLength}, \mathsf{id}^{a^{',j}}, m)$ and
   $(\mathsf{CheckLength}, \mathsf{id}^{a^j_{tr}}, l-m)$ in $\mathcal{F}_{\mathsf{CheckLength}}$ towards the verifier.

4. $\mathcal{S}$ then computes $y^j \leftarrow a^j - (2^m \cdot a^j_{tr} + a^{',j})$ using $\mathcal{F}_{\mathsf{CheckLength}}$ and then runs $(\mathsf{CheckZero}, \mathsf{id}^{y^j})$, which it makes output $(\mathsf{success})$.

The view of $\mathcal{V}^*$ simulated by $\mathcal{S}$ is distributed identically to its view in the real protocol. Any value being communicated to $\mathcal{V}^*$ is hidden in the commitment functionality.

□

D A Naïve Truncation Protocol

For comparison, we now describe a "naïve" way of truncating some value $[a]_M$ where $a \in [0, 2^l) \subset \mathbb{Z}_M$, without doing any conversions to $\mathbb{Z}_2$. Informally, the prover provides $[a]_M$ as well as its supposed bit decomposition $([a_0]_M, \ldots, [a_{l-1}]_M)$ authenticated in $\mathbb{Z}_M$. The prover then has to convince the verifier that each authenticated $[a_i]_M$ is a bit and that they all sum up to $[a]_M$, thus proving the correctness of the bit decomposition. Lastly, the prover and verifier can individually sum up most-significant $l-m$ bits, resulting in the truncated value $[a_{tr}]_M$.

Protocol $\Pi_{\mathsf{NaiveTrunc}}$

Input $[a]_M$ and it's supposed bit decomposition $([a_0]_M, \ldots, [a_{l-1}]_M)$.

Protocol

1. For $i = 0, \ldots, l-1$ compute $[y_i]_M = [a_i]_M \cdot (1 - [a_i]_M)$.
2. Let $[y]_M \leftarrow \sum_{i=0}^{l-1} [a_i]_M 2^i$.
3. Run $\mathsf{CheckZero}([y]_M, [y_0]_M, \ldots, [y_{l-1}]_M)$, output abort if the check fails and terminate.
4. Let $[a_{tr}]_M \leftarrow \sum_{i=0}^{l-m} [a_{l-m+i}]_M 2^i$.

Output $[a_{tr}]_M$.

Figure 22: Protocol that naïvely truncates a by m bits

This protocol is much more expensive than our `edaBit`-based approach, due to working in $\mathbb{Z}_M$ for all operations. Each bit must be committed to by a commitment over $\mathbb{Z}_M$, which itself requires $\log_2(M)$ bits of communication. Furthermore, the checking of each $[a_i]_M$ for $i \in [l]$ requires a multiplication, leading to further interaction. To give an example, we analyze the cost of this protocol when using Wolverine [WYKW20] to check the multiplications (alternative protocols such as [BMRS20] could also be used, but this does not significantly change the costs). For l multiplications in $\mathbb{Z}_M$, Wolverine runs a total of $(B-1) \cdot l$ iterations, each requiring 1 multiplication triples, for a total of $(3(B-1)) \cdot l$ random authentications and $(B-1)l$ fix (where fix corresponds to inputting a specific value into the commitment functionality) in $\mathbb{Z}_M$. Secondly, each iteration opens 2 values and performs a single $\mathsf{CheckZero}$. All calls to $\mathsf{CheckZero}$ may be batched together and performed at the end, but the other 2 must be done in each iteration, for a total of $l \cdot ((B-1) \cdot 2) + 1$ openings in $\mathbb{Z}_M$. Lastly, in step 3, all the checks for $a_i(1-a_i) \overset{?}{=} 0$ are batched together for a total of 1 opening. Throughout this analysis, we assume we're working in a small field such that $\log(M) \leq s$ for some security parameter. If instead it holds that $\log(M) > s$, then we can save a factor $(B-1)$ in these costs.

Table 4: Comparison of the costs of $\Pi_{\mathsf{NaiveTrunc}}$ (Figure 22) and $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 12).

	#Openings $\mathbb{F}_2$	#Openings $\mathbb{Z}_M$	#Faulty triples $\mathbb{F}_2$	#Faulty triples $\mathbb{Z}_M$
Naïve $\log(M) \leq s$	0	$l((B-1)\cdot 2)+2$	0	$(B-1)l$
Naïve $\log(M) > s$	0	$l \cdot 2 + 2$	0	l
Ours	$Bl + 2B$	$2B+1$	Bl	0
	#`(e)dabit` COTs	#`(e)dabit` VOLEs	#Bits from fix	
Naïve $\log(M) \leq s$	0	0	$2(B-1)l\log_2(M)$	
Naïve $\log(M) > s$	0	0	$2l \cdot \log_2(M)$	
Ours	$Bl + 2B$	$4B$	$(B+1)l + (4B+2)\log_2(M)$	

A breakdown of the costs of $\Pi_{\mathsf{NaiveTrunc}}$ compared to those of our optimized protocol $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 12) is given in Table 4, where we list both if $\log(M) \geq s$ but also $\log(M) > s$. In both cases, for typical parameters (e.g. $l = 32 \approx \log M$ and $B =$ 3–5) the naive protocol has much higher communication cost than ours, since the number of $\mathbb{Z}_M$ openings scales with the bit-length l. To give a concrete number, e.g. for the $\mathbb{Z}_p$ variant with $l = 32 \approx \log M$, when verifying a batch of around a million multiplications and 40-bit statistical security, we can use a bucket size $B = 3$. This leads to the communication of 8256 bits when using the naïve compared to only 960 when using ours, when we disregard the construction of the random authentications in $\mathbb{Z}_2$ and $\mathbb{Z}_p$ for both protocols.

E Sub-protocols

We look at the two sub-protocols convertBit2A and bitADDcarry that is used in our protocol verifying converion tuples.

E.1 Complexity of bitADDcarry

We assume that the input is distributed prior to running the protocol. The bitADDcarry circuit is implemented as a ripple-carry adder which computes the carry bit at every position with the following equation

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, \dots, m-1\} \tag{3}$$

where $c_0 = 0$ and x_i, y_i are the i'th bits of the two binary inputs. The output is then

$$z_i = x_i \oplus y_i \oplus c_i, \forall i \in \{0, \dots, m-1\} \tag{4}$$

and the last carry bit c_m. This requires m AND gates and as such m rounds of communication. As all the $\oplus$ can be computed by P_1 and P_2 locally (and as such requires no communication), 1 field element must be communicated per round. As this circuit is evaluated $B-1$ times per bucket, it results in a total for $(B-1)m$ field elements which must be communicated.

E.2 Complexity of convertBit2A

We consider the procedure convertBit2A as defined in Figure 6. We assume that the input (not the `daBit`) is distributed prior to running the protocol. This sub-protocol requires a single `daBit` to convert the bit authenticated in $\mathbb{F}_2$ to $\mathbb{F}_M$. Having a single `daBit` $([r]_2, [r]_M)$, we can convert a value $[x_m]_2$ by following the following protocol.

1. Compute $[c]_2 = [x_m]_2 + [r]_2$
2. $c \leftarrow \mathsf{Open}([c]_2)$

3. $[x]_M = c + [r]_M - 2 \cdot c \cdot [r]_M$.

We note that the only things requiring communication, is the distribution of the `daBit` used during the protocol and the opening of the value $[c]_2$. As such, we conclude that this requires the sending of four field elements (the opening of $[c]_2$ and the sending of the two bits of the `daBit`) and the cost of generating 1 `daBit`.

F Proofs of the $\mathbb{Z}_{2^k}$ Protocols

Here we present the full proofs of security that were omitted in Section 5.3.

F.1 Proof of Lemma 2

In the proof of Lemma 2, we make use of Lemma 1 from Cramer et al. [CDE+18] which we cite here:

Lemma 11 (Lemma 1 from [CDE+18])**.** *Let ℓ, r and m be positive integers such that $\ell - r \leq m$. Let $\delta_0, \delta_1, \dots, \delta_t \in \mathbb{Z}$, and suppose that not all the δ_i's are zero modulo 2^r, for $i > 0$. Let Y be a probability distribution on $\mathbb{Z}$. Then, if the distribution Y is independent from the uniform distribution sampling α below, we have*

$$\Pr_{\substack{\alpha, \chi_1, \dots, \chi_t \leftarrow_R \mathbb{Z}_{2^m}, \\ y \leftarrow_R Y}} \left[\alpha \cdot \left(\delta_0 + \sum_{i=1}^{t} \chi_i \cdot \delta_i \right) \equiv_\ell y \right] \leq 2^{-\ell + r + \log(\ell - r + 1)}.$$

Proof of Lemma 2. This proof is similar to the proof of Claim 1 in [CDE+18].

Since $(x_1, \dots, x_n) \not\equiv_k (0, \dots, 0)$, there is at least one i with $x_i \not\equiv_k 0$. Let $p' = p + \varepsilon \in \mathbb{Z}_{2^s}$ and $m' \in \mathbb{Z}_{2^{k+s}}$ denote the values send by the prover instead of p and m which are specified in the protocol. For $\mathcal{V}$ to accept, m' must satisfy the following equality:

$$m' \equiv_{k+s} 2^k \cdot \Delta \cdot p' + \sum_{i=1}^{n} \chi_i \cdot K[x_i] + 2^k \cdot K[r].$$

Subtracting (the honestly computed)

$$\begin{aligned} m &= \sum_{i=1}^{n} \chi_i \cdot M[x_i] + 2^k \cdot M[r] \\ &= \sum_{i=1}^{n} \chi_i \cdot (\Delta \cdot \tilde{x}_i + K[x_i]) + 2^k \cdot (\Delta \cdot \tilde{r} + K[r]) \\ &= 2^k \cdot \Delta \cdot p + \Delta \cdot \sum_{i=1}^{n} \chi_i \cdot x_i + \sum_{i=1}^{n} \chi_i \cdot K[x_i] + 2^k \cdot K[r] \end{aligned}$$

on both sides yields

$$m' - m \equiv_{k+s} 2^k \cdot \Delta \cdot \varepsilon - \Delta \cdot \sum_{i=1}^{n} \chi_i \cdot x_i \equiv_{k+s} \Delta \cdot \left(2^k \cdot \varepsilon - \sum_{i=1}^{n} \chi_i \cdot x_i \right).$$

Now, we invoke Lemma 1 from [CDE+18] (repeated as Lemma 11 on p. 41) with the following variables

$$\begin{array}{llll} \alpha := \Delta & \delta_0 := 2^k \cdot \varepsilon & \delta_i := -x_i & \chi_i := \chi_i \\ \ell := k + s & m := s & r := k & \end{array}$$

and get $2^{-\ell + r + \log(\ell - 1 + 1)} = 2^{-s + \log(s+1)}$ as upper bound on the success probability of $\mathcal{P}^*$ finding $m' - m$, which is necessary to make $\mathcal{V}$ output (success). □

F.2 Proof of Lemma 3

Proof of Lemma 3. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the $\mathsf{CheckMult}$ protocol with inputs as described in the lemma.

If the proposed multiplication triples $([x_i], [y_i], [z_i])_{i=1}^{\ell}$ are valid, i.e. $x_i \cdot y_i = z_i$ for $i = 1, \ldots, \ell$, and all commitments are opened to the correct values, then the values $w_k \neq 0$ for the invalid input triples due to the correctness of Beaver multiplication [Bea92]. So the verifier outputs $(\mathsf{failure})$.

Therefore, $\mathcal{P}^*$ has two possible options: 1. It can try to cheat during the $\mathsf{CheckZero}$ in Step 9 to reveal some different values $d', e' \neq d, e$ or $w_k \neq 0$ in Step 8. This succeeds with probability at most $\varepsilon_{\mathsf{cz}}$ (see Lemma 2). 2. It can choose to generate invalid multiplication triples. This can only be successful, if no invalid triples are detected in Step 7, and then invalid triples are paired up with invalid inputs in the right way. Weng et al. [WYKW20] have formalized this as a "balls and bins game". According to Lemma 2 of [WYKW20], an adversary wins this game with probability at most $\varepsilon_{\mathsf{cm}} = \binom{nB+C}{B}^{-1}$.

By the union bound, $\mathcal{P}^*$ can make $\mathcal{V}$ output $(\mathsf{success})$ with probability at most $\varepsilon_{\mathsf{cz}} + \varepsilon_{\mathsf{cm}}$. □

F.3 Proof of Theorem 4

Proof of Theorem 4. To show security in the UC-model, we construct a simulator $\mathcal{S}$ with access to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. The environment can choose to corrupt one of the parties, whereupon $\mathcal{S}$ simulates the interaction for the corrupted party. We cover the two cases separately, and first consider a corrupted prover, then a corrupted verifier.

Throughout the proof, we assume that the parties behave somewhat sensible, e.g. they use correct value identifiers, both parties access the functionality in a matching way, and that the simulator can always detect which method is to be executed.

Malicious Prover $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{P})$ to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. It also creates copies of the prover $\mathcal{P}^*$ and verifier $\mathcal{V}$, and runs the verifier according to the protocol $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$, while letting the prover behave as instructed by the environment. For this, $\mathcal{S}$ simulates the functionality of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ with corrupted $\mathcal{P}$. If the simulated $\mathcal{P}$ aborts the protocol, $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. The method calls are simulated as follows:

For Random, the parties call the Expand of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ to generate a commitment $[r]$ of the form $M[r] = \Delta \cdot \hat{r} + K[r]$. Since, $\mathcal{P}^*$ is corrupted, it is allowed to choose its outputs $\hat{r}, M[r] \in \mathbb{Z}_{2^{k+s}}$. $\mathcal{S}$ sends (Random) on behalf of $\mathcal{P}^*$ to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ and chooses $r := \hat{r} \bmod 2^k$ as value of the commitment. Hence, $\mathcal{P}^*$ receives $\hat{r}, M[r] \in_R \mathbb{Z}_{2^k+s}$ as in the real protocol (in the $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$-hybrid model). And $\mathcal{S}$ keeps track of all the commitments generated.

Affine is purely local, so there is no interaction to be simulated. $\mathcal{S}$ instructs the ideal functionality to perform the corresponding operations and computes the resulting commitments.

For $\mathsf{CheckZero}$, $\mathcal{S}$ first simulates the call to Random, and runs the protocol with the simulated parties. Then it sends the $\mathsf{CheckZero}$ message to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If the simulated verifier aborts, then $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$, which results in the ideal verifier aborting. To show that the verifier's output is indistinguishable between the real execution and the simulation we combine the following two facts: 1. If the verifier aborts in the real execution, then it does the same in the simulation. This holds by definition of the simulation. 2. If the verifier outputs $(\mathsf{success})$ in the real execution, then it does the same in the simulation except with probability at most $2^{-s+\log(s+1)}$. We show the contraposition, i.e. if the verifier aborts in the simulation, then it does the same in the real execution except with the given probability. By definition of $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$, the premise hold if one of the input commitments contains a non-zero value. Thus, we can apply Lemma 2, which gives us the desired consequence.

For Input, the parties first invoke Random to obtain a commitment $[r]$, so $\mathcal{S}$ simulates this (see above). Input is the only method, where the prover has a private input. The simulator can extract it from $\mathcal{P}^*$'s message $\delta \in \mathbb{Z}_{2^k}$ by computing $x \leftarrow \delta + r$ (it knows r because it simulates the $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ functionality). Then

$\mathcal{S}$ can send (Input, x) on behalf of the corrupted prover to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. For correctness, note that a commitment $[r] + (x - r)$ contains the value x iff. $[r]$ is a commitment to r.

Since Open is implemented in terms of Affine and $\mathsf{CheckZero}$, and we have that a commitment $[x]$ contains a value x iff. $[x] - x$ is a commitment to 0. We can simulate the methods as describe above. Hence, the simulation of Open fails exactly if the simulation of $\mathsf{CheckZero}$ fails.

$\mathsf{CheckMult}$ is simulated in the same way as $\mathsf{CheckZero}$. Here, we apply Lemma 3, and get that the output of $\mathcal{V}$ is the same in the simulation and in the real execution except with probability at most $\varepsilon_{\mathsf{cz}} + \varepsilon_{\mathsf{cm}}$.

This concludes the proof for the case of a corrupted prover. As shown above, we can simulate its view perfectly for all methods. Overall, by the union bound, the environment has an distinguishing advantage of

$$(q_{\mathsf{cz}} + q_{\mathsf{cm}}) \cdot \varepsilon_{\mathsf{cz}} + q_m \cdot \varepsilon_{\mathsf{cm}}.$$

Malicious Verifier The setup of the simulation in case of a corrupted verifier $\mathcal{V}^*$ is similar as before. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. It creates copies of the prover $\mathcal{P}$ and verifier $\mathcal{V}^*$. The prover is run according to the protocol, whereas the environment controls the verifier. For this, $\mathcal{S}$ simulates the functionality of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ with corrupted $\mathcal{V}$. For all methods, since $\mathcal{V}$ does not have any private inputs no input extraction is necessary. So the simulator can just send the corresponding message on behalf of the verifier to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. The method calls are simulated as follows:

During initialization, $\mathcal{S}$ allows $\mathcal{V}^*$ to choose its MAC key Δ with the simulated $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ functionality.

For Random, the parties call the Expand of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ to generate a commitment $[r]$ of the form $M[r] = \Delta \cdot \hat{r} + K[r]$ where $\mathcal{V}^*$ can choose $K[r]$. $\mathcal{S}$ sends (Random) on behalf of $\mathcal{V}^*$ to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$.

As before, Affine is purely local, so there is no interaction to be simulated. $\mathcal{S}$ instructs the ideal functionality to perform the corresponding operations and computes the resulting commitments.

For $\mathsf{CheckZero}$, $\mathcal{S}$ sends the respective message to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$, which finishes the simulation. Otherwise, $\mathcal{S}$ simulates the normal protocol execution: It first simulates the call to Random and lets $\mathcal{V}^*$ choose the coefficients χ_i. Since $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ did not abort, we know that $x_1 = \cdots = x_n = 0$. We also know $\Delta, K[x_1], \ldots, K[x_n], K[r]$, so we can sample $p \in_R \mathbb{Z}_{2^s}$ and compute $m := 2^j \cdot \Delta \cdot p + \sum_{i=1}^{n} \chi_i \cdot K[x_i] + 2^k \cdot K[r]$ as expected by the verifier.

For Input, $\mathcal{S}$ first simulates the call to Random as above, and then sends a random value $\delta \in_R \mathbb{Z}_{2^k}$ to the simulated verifier. Also, $\mathcal{S}$ sends (Input) on behalf of $\mathcal{V}^*$ to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$.

For Open, $\mathcal{S}$ sends the Open on behalf of $\mathcal{V}^*$ to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ and receives the committed values $x_1, \ldots, x_n \in \mathbb{Z}_{2^k}$ as output. It sends these values to the simulated verifier, and then simulates Affine and $\mathsf{CheckZero}$ as above. So the view is distributed identically to the real protocol.

For $\mathsf{CheckMult}$, $\mathcal{S}$ sends the corresponding message on behalf of the corrupted verifier to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$. Otherwise, $\mathcal{S}$ simulates the complete protocol using the constant value 0 for all of the prover's commitments. Because the simulated $\mathcal{P}$ behaves like an honest prover, it samples all multiplication triples $([x_i], [y_i], [z_i])_{i=1}^{\ell}$ correctly. Since the view of the $\mathcal{V}$ is distributed identically to the real execution and independent of the prover's real inputs: The opened triples in Step 6 are uniformly distributed, valid multiplication triples. The values d, e revealed in Step 8a are distributed uniformly in $\mathbb{Z}_{2^k}$, and the $\mathsf{CheckZero}$ passes since the w_k are all 0.

This concludes the proof for the case of a corrupted verifier. As shown above, we can simulate its view perfectly for all methods. Overall, the environment has a distinguishing advantage as stated in the theorem. □

F.4 Proof of Theorem 5

Proof of Theorem 5. Since most of $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ is actually identical to $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ we will refer to the Proof of Theorem 4 for these parts, and focus on the differences here.

The subroutines $\mathsf{CheckZero}$ and $\mathsf{CheckZero}'$ are only very slightly modified from the $\mathsf{CheckZero}$ from $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$. The latter is exactly the same as in before, but for the larger message space $\mathbb{Z}_{2^{k+s}}$, and the former additionally hides some more bits. Hence, the same Lemma 2 can be applied here.

The remaining part of the proofs considers the different implementation of $\mathsf{CheckMult}$:

Malicious Prover The setup of the simulation is the same as in the Proof of Theorem 4, i.e. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{P})$ to the ideal functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ and simulates copies of prover and verifier.

For the method $\mathsf{CheckMult}$, $\mathcal{S}$ can exactly simulate the protocol since it knows all the commitments, and η is sampled uniformly at random from $\mathbb{Z}_{2^s}$.

If the simulated verifier aborts, it sends (abort) to $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. Thus, if the verifier aborts in the real execution, then it does the same in the simulation. On the other hand, if the verifier aborts in the simulation, then by Lemma 4 it also aborts in the real protocol, except with probability $\varepsilon_{\mathsf{cz}} + \varepsilon'_{\mathsf{cm}}$.

Malicious Verifier Again, we have the same setup as before, i.e. the simulator sends $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ and simulates copies of prover and verifier.

For $\mathsf{CheckMult}$, we use the same strategy as in the Proof of Theorem 4: $\mathcal{S}$ sends the corresponding message on behalf of the corrupted verifier to $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$. Otherwise $\mathcal{S}$ simulates the complete protocol using the constant value 0 for all of the prover's commitments so that the verifier's view is the same as in the real execution.

Summarizing, we have shown that no environment can distinguish the simulation from a real execution of the protocol with more than the stated advantage. $\square$