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

Carsten Baum  Lennart Braun  Alexander Munch-Hansen  Peter Scholl  
Aarhus University  Aarhus University  Aarhus University  Aarhus University  
cbaum@cs.au.dk  braun@cs.au.dk  almun@cs.au.dk  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 [BCGHS, 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 \( x = \sum_{i=0}^{m-1} 2^i x_i \mod M \).

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 \( P \) and verifier \( V \):

1. \( P \) commits to \([x_0]_2,\ldots,[x_{m-1}]_2\).
2. \( P \) and \( V \) run the edaBits protocol to generate a valid committed edaBit \(((r_0)_2,\ldots,[r_{m-1}]_2,[r]_M)\).
3. \( P \) 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, \( P \) will first commit to \([x]_M\), then open \([x+r]_M\) to \( V \), 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 ∈ ZM is at most m bits in length, for some public m. We then show that integer truncation in ZM 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 < y), where [x]_M, [y]_M are committed.

ZK for Arithmetic Circuits over Z_{2^k}. Our conversion, truncation and comparison protocols can all be made to work with either a field Z_p, or a ring Z_{2^k}, giving flexibility in high-level applications. While ZK protocols for Z_p and Z_2 have been well-studied, there is less work on protocols for circuits over 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 Z_{2^k}, by adapting the techniques from SPDZ_{2^k} [CDE+18]. Given the MACs, which serve as homomorphic commitments in 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 [BCG18]). Removing the trusted setup can be done with an actively secure VOLE protocol over Z_{2^k}. We note that the LPN-based construction of [BCG18] also works over 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 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 $P$ to convince a verifier $V$ that a certain statement is true (and that it possesses a witness to this fact). This happens in a way such that $V$ 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 $F_{\text{ZK}}$ for satisfiability of circuits $C$: On input $(\text{Prove}, C, w)$ from $P$ and $(\text{Prove}, C)$ from $V$ the functionality $F_{\text{ZK}}$ outputs $\top$ to $V$ iff. $C(w) = 1$ holds, and sends $\bot$ otherwise [WYKW20].

Following previous works (e.g. [WYKW20] [BMRS20]), we use the commit-and-prove strategy to instantiate $F_{\text{ZK}}$ using homomorphic commitments (see Section 2.4). These allow the prover $P$ to commit to its witness $w$. Then the circuit $C$ can be evaluated on the committed witness to obtain a commitment to the output, which is opened to prove that indeed $C(w) = 1$ 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 OT 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 \( \beta \) are vectors of the same length rather than single field elements. Subfield VOLEs [BCG+19b] furthermore extends this concept such that the elements of \( \alpha \) and \( \beta \) live in an extension field \( \mathbb{F}_{p^r} \supseteq \mathbb{F}_p \). 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 \( \Delta, K[x_1], \ldots, K[x_n] \in \mathbb{Z}_p \) are chosen by \( \mathcal{V} \), and then \( \mathcal{P} \) obtains the MACs \( M[x_i] = \Delta \cdot x_i + K[x_i] \in \mathbb{Z}_p \). We use the notation \([x_i]_p\) for this. To open \([x_i]_p\), \( \mathcal{P} \) 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 \( \Delta \) 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 \( \mathcal{F}^R_{\text{ComZK}} \) 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, \( \mathcal{F}^R_{\text{ComZK}} \) 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 simplify the presentation of higher-level protocols without explicitly mentioning the commitment identifiers. We use also shorthands for the different methods of \( \mathcal{F}^R_{\text{ComZK}} \), 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 $F_{\text{ComZK}}^R$**

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

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

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

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

**Affine Combination** On input (Affine, id$_0$, id$_1$, \ldots, id$_n$, $a_0$, \ldots, $a_n$) from $P, V$ where $(id_i, x_{id_i}) \in st$ for $i = 1, \ldots, n$ and $(id_n, \cdot) \notin st$:

1. Set $x_{id_0} \leftarrow a_0 + \sum_{i=1}^{n} a_i \cdot x_{id_i}$ and $st \leftarrow st \cup \{(id_0, x_{id_0})\}$.

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

**CheckZero** On input (CheckZero, id$_1$, \ldots, id$_n$) from $P, V$ and where $(id_i, x_{id_i}) \in st$ for $i = 1, \ldots, n$:

1. If $x_{id_1} = \cdots = x_{id_n} = 0$, then send (success) to $V$, otherwise send (abort) to all parties and terminate.

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

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

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

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

**Open** On input (Open, id$_1$, \ldots, id$_n$) from $P, V$ where $(id_i, x_{id_i}) \in st$ for $i = 1, \ldots, n$:

1. Send $x_{id_1}, \ldots, x_{id_n}$, to $V$.

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

**MultiplicationCheck** Upon $P \ & \ V$ inputting (CheckMult, (id$_{x_i}$, id$_{y_i}$, id$_{z_i}$)$_{i=1}^{n}$) where $(id_{x_i}, x_i), (id_{y_i}, y_i), (id_{z_i}, z_i) \in st$ for $i = 1, \ldots, n$:

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

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

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

6
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. $([b_0], \ldots, [b_{m-1}][b]_M)$, 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 $([x_0], \ldots, [x_{m-1}][x]_M)$ is indeed consistent, i.e. that $x = \sum_{i=0}^{m-1} x_i 2^i \mod M$.

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 $r_j = \sum_{i=0}^{m-1} 2^i r_{i,j}$ 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 $([x_0], \ldots, [x_{m-1}][x]_M)$ online. Instead, the prover can commit to $([x_0], \ldots, [x_{m-1}][x]_M)$ 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 $([x_0], \ldots, [x_{m-1}][x]_M)$ is called a conversion tuple. We call $([x_0], \ldots, [x_{m-1}][x]_M)$ consistent iff $x = \sum_{i=0}^{m-1} 2^i x_i \mod M$. 


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 $F_{\text{ComZK}}^{R}$ in Figure 2, which extends two instances of $F_{\text{ComZK}}^{R}$ for $R = \mathbb{Z}_2$ and $R = \mathbb{Z}_M$ and simply parses all method calls to the respective instance.

**Functionality $F_{\text{Conv}}^{2,M}_{\text{ComZK}}$**

$F_{\text{ComZK}}^{2,M}$ communicates with two parties $P, V$. It contains two separate instances of the commitment functionality $F_{\text{ComZK}}^{R}$, 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 $F_{\text{ComZK}}^{R}$ 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 $F_{\text{ComZK}}^{R}$.

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

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

**Functionality $F_{\text{Conv}}$**

$F_{\text{Conv}}$ extends the existing functionality $F_{\text{ComZK}}^{2,M}$, 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 $P$ and $V$ inputting $(\text{VerifyConv}, N, m, \{(id^{(j)}_0, \ldots, id^{(j)}_{m-1}), \{id^{(j)}_j\}_{j \in [N]}))$:

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

Figure 3: Functionality $F_{\text{Conv}}$ checking edaBits

### 3.2 The Conversion Verification Protocol $\Pi_{\text{Conv}}$

The following protocol $\Pi_{\text{Conv}}$ verifies the correctness of a batch of $N$ conversion tuples. $\Pi_{\text{Conv}}$ uses $F_{\text{Dabit}}$ (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.
The two properties hold:

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

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 $\{(r_0)_2, \ldots, [r_{m-1}]_2, [r]_M\}_{j=0}^{B-1}$. 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 $([c_0]_2, \ldots, [c_m]_2) = ([c_0]_2, \ldots, [c_{m-1}]_2) + ([r_0]_2, \ldots, [r_{m-1}]_2)$. 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 $C : \mathbb{Z}_2^{2m} \rightarrow \mathbb{Z}_2^{m+1}$ 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 evaluate 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 $\Pi_{\text{Conv}}$

Assume that $F_{\text{Dabit}}$ contains $N$ committed conversion tuples $\{[c_0^{(i)},...,c_{m-1}^{(i)}], [e_{i}]_M \}_{i \in [N]}$.

// $P$ commits auxiliary values for conversion check. daBits are then verified.

1. $P$ commits to the following values using $F_{\text{Dabit}}$:
   (a) Random $\text{edaBits} ([r_0^{(j)},...,r_{m-1}^{(j)}], [r^{(j)}]_M)_{j \in [NB+C]}$.
   (b) Random $\text{daBits} ([b^{(j)}], [b^{(j)}]_M)_{j \in [NB]}$.
   (c) Random multiplication triples
       $([z^{(j)}], [z^{(j)}]_2)_{j \in [NBm+Cm]}$

2. $P$ and $V$ send $(\text{VerifyDabit}, NB, \{[b^{(j)}], [b^{(j)}]_M\})_{j \in [NB]}$ to $F_{\text{Dabit}}$.

// $P$ and $V$ shuffle the auxiliary values and a subset gets opened and verified.

3. $V$ 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 $P$.

4. Both parties shuffle the $\text{edaBits} [r_0^{(j)},...,r_{m-1}^{(j)}], [r^{(j)}]_M$ locally according to $\pi_1$. They then shuffle $[b^{(j)}], [b^{(j)}]_M$ according to $\pi_2$ and $[z^{(j)}], [z^{(j)}]_2$ according to $\pi_3$.

5. Run a cut-and-choose procedure as follows:
   (a) $P$ opens $\{[r_0^{(j)},...,r_{m-1}^{(j)}], [r^{(j)}]_M\}_{j \in [NB+C]}$ (the last $C \text{ edaBits}$) towards $V$, who in turn checks that $r^{(j)} = \sum_{i=0}^{m-1} 2^i \cdot r_i^{(j)}$.
   (b) $P$ opens the $x, y$ values for the last $Cm$ triples $\{[x^{(j)}], [y^{(j)}]_2\}_{j \in [NBm+Cm]}$ and proves to $V$ that $\text{CheckZero}([z^{(j)}] - x^{(j)} \cdot y^{(j)})$ for all opened triples.

// $P$ and $V$ verify each conversion tuple in a bucket.

6. For the $i$'th conversion tuple $[c_0,....,c_{m-1}, c]_M$, do the following for $j \in [B]$:
   (a) Let $[r_0,....,r_{m-1}, r]_M$ be the $(i-1) \cdot B + j$'th $\text{edaBit}$ and $[c + r]_M = [c]_M + [r]_M$.
   (b) Let $([c_0],....,[c_{m-1}], [r_0],....,[r_{m-1}]) \leftarrow \text{bitADDcarry}([c_0],....,[c_{m-1}], [r_0],....,[r_{m-1}])$.
   (c) Convert $[c]_M \leftarrow \text{convertBit2A}([c_0]_2)$ using the $(i-1) \cdot B + j$'th $\text{daBit} ([b_0], [b]_M)$.
   (d) Let $[e']_M \leftarrow [c + r]_M - 2^m \cdot [c]_M$.
   (e) Let $e_i \leftarrow \text{Open}([e_i]_2)$ for $i = 0,...,m-1$. Then run $\text{CheckZero}([e']_M - \sum_{i=0}^{m-1} 2^{i} \cdot e_i)$.

7. If any of the checks fail, $V$ outputs $\text{abort}$. Otherwise it outputs $\text{success}$.

Figure 5: Protocol $\Pi_{\text{Conv}}$ to verify Conversion Tuples
care must be taken regarding the carry from \( m \) (which "hides"
Instead, we simply ignore the carry bit and set
Remark 1. When
Due to space constraints, the full proof of security can be found in Appendix A. We summarize the proof
3.3 Proof of security
While \texttt{bitADDcarry} will ensure that (assuming correct triples) \(([, e_0], \ldots, [e_m])\) 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 \texttt{daBit} to convert \([e_m]\) 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 \(([e_0], \ldots, [e_m])\) and check that \( e' = \sum_{i=0}^{m-1} 2^i \cdot e_i \).

Remark 1. When \( M = 2^k \), we can optimize \( \Pi_{\text{Conv}} \) by removing the conversion step 6(d), which uses \texttt{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 \( \Pi_{\text{Conv}} \), 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 \( \Pi_{\text{Conv}} \). 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 \( K \cdot B \ \texttt{edaBits} \) 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 \texttt{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 \texttt{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:

**Procedure convertBit2A**

**Input** A daBit \(([, r], [r])\) and a commitment \([x]\).

**Protocol**

1. \( c \leftarrow \text{Open}([r]_2 \oplus [x]_2) \).
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 \([x_0], \ldots, [x_{m-1}], [y_0], \ldots, [y_{m-1}]\).

**Protocol** Let \( c_0 = 0 \).

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

Figure 7: A ripple-carry adder.
Theorem 1. The probability of \( \Pi_{Conv} \) 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 \( \Pi_{Conv} \):

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

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 \( ([b]_2,[b]^\prime]_M) \), where \( b \) is a random bit. Generating consisting daBits requires verifying that \( [b]^\prime \) 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 \( \Pi_{Conv} \) 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 \( b \equiv b' \mod 2 \), but not necessarily \( b' \in \{0,1\} \).

In Step 6 of \( \Pi_{Conv} \) (Figure 3), daBits are used in convertBit2A (Figure 6) to transform the final carry bit \( [e^\prime]_M \) from bitADDcarry (Figure 7) into \( [e']_M \) such that \( e^\prime = e' \). 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 \( e^\prime = e^\prime + \delta \cdot |e^\prime| \) where \( \delta = (-1)^{b-b'} \) where \( (b-b') > 1 \) represents the difference (or error) between \( b' \) and \( b \). The result, \( |e^\prime| > 1 \) where \( |\cdot| \) denotes absolute value. This carry bit \( [e^\prime]_M \) is then used to remove the potential carry from \([c+r]_M \) by computing

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

in Step 6d of \( \Pi_{Conv} \). However, when \( [e^\prime]' \not\in \{0,1\} \) 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 \( \Pi_{Conv} \), can be found in Appendix [B].

Formally, we define an ideal functionality \( \mathcal{F}_{FDabit} \) (Figure 20) in Appendix [B] that encompasses this idea of two bits \( (b,b') \) such that \( b \equiv b' \mod 2 \).

Creating Faulty daBits We now describe a revised daBit verification protocol \( \Pi_{FDabit} \) realizing \( \mathcal{F}_{FDabit} \), that, for a given daBit \( ([b]_2, [b]'_M) \), only verifies that \( b \equiv b' \mod 2 \). 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 \( \Pi_{FDabit} \) in Figure 8 and show the following statement in Appendix [B]:

Lemma 1. Protocol \( \Pi_{FDabit} \) (Figure 8) UC-realizes \( \mathcal{F}_{FDabit} \) (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
Table 1: Costs of verifying a conversion between $Z_p$ and $Z_m$. 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 $\gamma$ random bits $\{[c_i]_p \mid i \in [\gamma]\}$. It additionally creates $[c_1]_2$. It does not prove consistency among $[c_1]_p$ and $[c_1]_2$.
2. $\mathcal{P}$ shows that $\{[c_i]_p \mid i \in [\gamma]\}$ are bits by showing that CheckZero([c_i]_p \cdot (1 - [c_i]_p)) = (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. $r \leftarrow \text{Open}([r]_2)$
6. Let $[r']_M \leftarrow \left(\sum_{i=1}^{N} e_i \cdot [b_i]_p\right)$
7. Let $\tau = \text{Open}([r']_p + \sum_{j=1}^{\gamma} [c_i]_p \cdot 2^{j-1})$.
8. Check if $0 \leq \tau < 2^\gamma$ and $r = \tau \mod 2$. If not, abort.

Output (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 $Z_M$ 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 bitADDCarry and convertBit2A.

Table 1: Costs of verifying a conversion between $Z_p$ and $Z_m$ (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. $m \leq \lceil \log(p) \rceil$, $k$ denotes the bitsize of the converted value, and $B$ is the bucket size.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Comm. in bits</th>
<th>#COTs</th>
<th>#VOLEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic, $Z_p$, $\log(p) \leq s$</td>
<td>$13Bm + 6m + B - 1$</td>
<td>$4Bm + 3m + B - 1$</td>
<td>$11B - 4$</td>
</tr>
<tr>
<td>basic, $Z_p$, $\log(p) &gt; s$</td>
<td>$10Bm + 6m + B - 1$</td>
<td>$4Bm + 3m + B - 1$</td>
<td>$8B - 4$</td>
</tr>
<tr>
<td>optimized, $Z_p$</td>
<td>$6Bm + B$</td>
<td>$4Bm + B$</td>
<td>$2B$</td>
</tr>
<tr>
<td>optimized, $Z_{2^k}$</td>
<td>$5Bm$</td>
<td>$4Bm$</td>
<td>$B$</td>
</tr>
</tbody>
</table>

The “basic” baseline comparison in Table 1 comes from a straightforward application of using edaBits for
ZK, similarly to \cite{EGK+20}. Namely, this protocol would first generate consistent \emph{edaBits} using \cite{EGK+20}, and then verify the conversion using a single binary addition circuit (similar to the bucket-check in Figure 3 step 4). However, this requires doing the check with \emph{m} verified multiplication triples (over \(\mathbb{Z}_2\)) and a single \emph{daBit}, which in turn requires an additional verified multiplication (over \(\mathbb{Z}_{M}\)). To estimate these costs, we used the Wolverine \cite{WYKW20} protocol for verifying AND gates at a cost of 7 bits per gate, and Mac’N’Cheese \cite{BMRS20} for verifying triples in a larger field.

Since COTs and VOLEs can be obtained from pseudorandom correlation generators with very little communication \cite{YWL20, 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 \cite{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.

\section{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 naive 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 \emph{edaBits} in addition to possibly other operations. Instead, we only have to verify that the input forms a consistent \emph{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 \(\mathbb{Z}_M\).

\subsection{Truncation}

In Figure 9 we present a functionality \(F_{\text{VerifyTrunc}}\) that takes a batch of commitments \([a_j]_M\) and their supposed truncations (by \(m\) bits) \([a’_j]_M\). The functionality ensures that the truncations are correct, namely, \(a’_j = \lfloor \frac{a_j}{2^m}\rfloor\). 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 \(F_{\text{VerifyTrunc}}\) using just a single call to our batch conversion protocol, \(F_{\text{Conv}}\), 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 \(([a_{tr}]_2, \ldots, [a_{tr}^{l-m-2}]_2\))
- 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 \( \Pi_{\text{Conv}}\) on \([a_{tr}]_M\) and \(([a’_{tr}]_2, \ldots, [a’_{tr}^{l-m-2}]_2\)) not only shows consistency
The protocol $\Pi_{\text{VerifyTrunc}}$ also directly follows from the consistency check described in Figure 5. The output of the comparison. We remark that, similarly to previous works in the MPC setting [Cd10, EGK +20], the two representations of $\alpha$, $\beta$ bits (same goes for $\alpha^0, \beta^0$) a set of integers $\{m^j\}_{j \in [N]}$ such that $m^j \in [M]$ represents the how much $a^0, \beta^0$ is truncated to reach $a^1, \beta^1$ 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 $P$ and $V$ inputting $(\text{VerifyTrunc}, N, \{m^j, (id^0, id^1)\}_{j \in [N]})$:

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

Figure 9: Functionality $F_{\text{VerifyTrunc}}$ 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 $F_{\text{CheckLength}}$ (Figure 10) that encapsulates this concept of using $F_{\text{Conv}}$ as a way of bounding the size of an authenticated value.

The protocol $\Pi_{\text{CheckLength}}$ 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 $F_{\text{Conv}}$. The cost of the protocol also directly follows from the consistency check described in Figure 4.

We prove that $\Pi_{\text{VerifyTrunc}}$ securely realises the functionality $F_{\text{VerifyTrunc}}$, in Appendix C. Note that $\Pi_{\text{CheckLength}}$ and $\Pi_{\text{VerifyTrunc}}$ do not utilise anything specific about $M$ except $l \leq M$ and both work for $Z_p$ and $Z_{2^k}$.

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

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

Figure 10: Functionality to verify length of commitments

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

### 4.2 Integer Comparison

We now discuss how to compare two signed, $l$-bit integers $\alpha$ and $\beta$. 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 $\Pi_{\text{VerifyTrunc}}$ 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 \( \Pi_{\text{CheckLength}} \)**

**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 \( j \in [N] \), \( P \) commits to \([x^j_0]^2, \ldots, [x^j_{m^j-1}]^2\).
2. Let \( m = \max_j \{ m^j \} \), and for \( i = m^j, \ldots, m - 1 \), let \([x^j_i]^2\) denote a dummy commitment to zero (which can be easily obtained with \( \text{CheckZero} \)).
3. Run \( \mathcal{F}_{\text{Conv}} \) on \( \{ ([x^j_0]^2, \ldots, [x^j_{m^j-1}]^2, [x^j]^M) \}_{j \in [N]} \) and output what \( \mathcal{F}_{\text{Conv}} \) outputs.

Figure 11: Protocol \( \Pi_{\text{CheckLength}} \) that verifies that committed elements are bounded.

**Protocol \( \Pi_{\text{VerifyTrunc}} \)**

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

**Protocol**

1. For each \( j \in [N] \), \( P \) commits to the least-significant \( m \) bits of \([a^j]^M \), denoted as \([a^j']^M = [a^j \mod 2^m]^M\).
2. The parties call \( \mathcal{F}_{\text{CheckLength}} \) with input \( \{ [a^j']^M, m^j \}, j \in [N] \cup \{ [a^j_{tr}]^M, l - m^j \}, j \in [N] \).
3. For each \( j \), let \([y]^M = [a^j]^M - (2^m \cdot [a^j_{tr}]^M + [a^j']^M) \) and run \( \text{CheckZero}([y]^M) \).
   Abort if any of the checks fail. Otherwise output (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 \( \mathcal{F}_{\text{ComZK}}^{\mathbb{Z}_{2^k}} \), 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 SPDZ\( _{2^k} \) [CDE+18]: Let \( s \) be a statistical security parameter. To authenticate a value \( x \in \mathbb{Z}_{2^k} \) known to \( P \) towards \( V \) (denoted as \([x] \)), we choose the MAC keys \( \Delta \in_R \mathbb{Z}_{2^s} \) and \( K[x] \in_R \mathbb{Z}_{2k+s} \), and compute the MAC tag as

\[
M[x] := \Delta \cdot \bar{x} + K[x] \in \mathbb{Z}_{2k+s}, \tag{1}
\]
where \( x = \tilde{x} \mod 2^k \), i.e. \( \tilde{x} \) is a representative of the corresponding congruence class of integers modulo \( 2^k \). Then \( P \) gets \( \tilde{x} \) and \( M[x] \), whereas \( V \) receives \( \Delta \) and \( K[x] \).

Initially \( \tilde{x} \) may be chosen as \( \tilde{x} = x \in \{0, \ldots, 2^k - 1\} \). 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 \( P \).

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

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

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 SPDZ_{2^k}, the MACs are obtained using vector OLE over rings. We describe the protocols in the \( F_{\text{vole2k}}^{s,r} \)-hybrid model (cf. Figure 13). This functionality can be instantiated as in \( CDE+18 \), e.g. from the oblivious transfer protocol of \( \text{Sch18} \). 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 \( r \in R \mathbb{Z}_{2^k} \). 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 \( \Pi_{\text{ComZK-a}}^{Z\mathbb{Z}_{2^k}} \) (Figures 14 & 15).

### Vector Linear Oblivious Evaluation for \( \mathbb{Z}_{2^k} \): \( F_{\text{vole2k}}^{s,r} \)

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

1. If \( V \) is honest, it samples \( \Delta \in_R \mathbb{Z}_{2^k} \) and sends \( \Delta \) to \( V \).
2. If \( V \) is corrupt, it receives \( \Delta \in \mathbb{Z}_{2^k} \) from \( S \).
3. \( \Delta \) 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 \( V \) is corrupted, it receives \( K[x] \in \mathbb{Z}_{2^r} \) from \( S \) instead.
3. If \( P \) is corrupted, it receives \( x, M[x] \in \mathbb{Z}_{2^r} \) from \( S \) instead, and computes \( K[x] \leftarrow M[x] - \Delta \cdot x \in \mathbb{Z}_{2^r} \).
4. \((x, M[x])\) is sent to \( P \) and \( K[x] \) is sent to \( V \).

Figure 13: Ideal functionality for Vector-OLE with key size \( s \) and message size \( r \). The functionality is based on \( F_{\text{vOLE}}^{s,r} \) from Wolverine [WYKW20] Fig. 2.
5.2 Instantiation of $\mathcal{F}^{Z_{2^k}}_{\text{ComZK}}$

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

$\Pi^{Z_{2^k}}_{\text{ComZK-a}}$ (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 $\ell$ 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 $(a_i \cdot b_i \equiv c_i)$ 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 $\Pi^{Z_{2^k}}_{\text{ComZK-b}}$ (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 $Z_{2^k}$-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 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 $k$ bits, which might lead to additional problems. We overcome these challenges by further increasing the ring size from $Z_{2^{k+s}}$ to $Z_{2^{k+s+1}}$, 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 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 $\Pi^{Z_{2^k}}_{\text{ComZK-a}}$

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

**Theorem 4.** The protocol $\Pi^{Z_{2^k}}_{\text{ComZK-a}}$ (Figures 14 & 15) securely realizes the functionality $\mathcal{F}^{Z_{2^k}}_{\text{ComZK}}$: No environment can distinguish the real execution from a simulated one except with probability $(q_{cz} + q_{cm}) \cdot 2^{-s + \log(s + 1)} + q_{cm} \cdot \binom{nB + C}{B}^{-1}$, where $q_{cz}$ is the sum of calls to CheckZero and Open, and $q_{cm}$ the number of calls to 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 the soundness properties of the CheckZero and CheckMult protocols in $\Pi^{Z_{2^k}}_{\text{ComZK-a}}$. These are stated in the following two lemmata. The full proof of Theorem 4 is given in Appendix F.3.

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


Analysis also applies to the \( Z \).

### 3.2 Proof of \( \Pi \)

**Lemma 2.**

and inputs \( (\varepsilon, \ldots, n) \) with probability at most 2.

**Lemma 3.**

It is given in Appendix F.1.

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

### 5.3.2 Proof of \( \Pi \)

The formal statement of security is given in the following theorem:
Lemma 4. If \( \text{operation} \), that we use to show indistinguishability of our simulation. The proof is included here, since it

\[ \epsilon \]

Lemma 2 again: Hence, a

\[ P \]

Suppose

\[ \epsilon \]

Proof.

\[ \epsilon \]

\[ V \]

Figure 15: Protocol \( \Pi_{\text{ComZK-a}} \) of MultiplicationCheck

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

1. \( P \) aborts if \( a_i \cdot b_i \neq c_i \) (mod \( 2^k \)) for some \( i = 1, \ldots, n \).
2. Let \( \ell := n \cdot B + C \), and initialize \( \text{lst} \leftarrow \emptyset \)
3. They compute \( ([x_i], [y_i])_{i=1}^\ell \leftarrow \text{Random}() \) so that \( P \) receives \( (x_i, y_i)_{i=1}^\ell \).
4. \( P \) computes \( z_i \leftarrow x_i \cdot y_i \) for \( i = 1, \ldots, \ell \), and they run \( ([z_i])_{i=1}^\ell \leftarrow \text{Input}((z_i))_{i=1}^\ell \).
5. \( V \) samples a permutation \( \pi \in \mathcal{R} \) and sends it to \( P \).
6. They run \( (x_{\pi(i)}, y_{\pi(i)}, z_{\pi(i)})_{i=1}^C \leftarrow \text{Open}(([x_{\pi(i)}], [y_{\pi(i)}], [z_{\pi(i)}])_{i=1}^C, \text{lst}) \).
7. \( V \) checks if \( x_{\pi(i)} \cdot y_{\pi(i)} = z_{\pi(i)} \) for \( i = 1, \ldots, C \), and aborts otherwise.
8. For each \( (a_j, b_j, c_j) \) with \( j = 1, \ldots, n \) and for each \( (x_{\pi(k)}, y_{\pi(k)}, z_{\pi(k)}) \) with \( k = C + (j - 1) \cdot B + 1, \ldots, C + j \cdot B \), they compute
   
   - \( a \leftarrow \text{Open}([a_j] - [x_{\pi(k)}], \text{lst}) \)
   - \( c \leftarrow \text{Open}([y_{\pi(k)}], \text{lst}) \)
   - \( b \leftarrow \text{Open}([b_j] - [y_{\pi(k)}], \text{lst}) \)
   - \( d \leftarrow \text{Open}([b_j] - [y_{\pi(k)}], \text{lst}) \)
   - \( e \leftarrow \text{Open}([b_j] - [y_{\pi(k)}], \text{lst}) \)

9. Finally, they run \( (\text{CheckZero}, \text{lst}, ([w_j])_{j=C+1}^n) \). If successful and the check in Step 7 also passed, \( V \) outputs (success) and aborts otherwise.

![Protocol \( \Pi_{\text{ComZK-a}} \) (Part 2)](image)

**Theorem 5.** The protocol \( \Pi_{\text{ComZK-b}} \) (Figure 16) securely realizes the functionality \( F_{\text{ComZK}} \): No environment can distinguish the real execution from a simulated one except with probability \( (\eta_{\text{zx}} + \eta_{\text{cm}}) \cdot 2^{-\alpha_{\text{cm}}} + \eta_{\text{cm}} \cdot 2^{-s} \), where \( \eta_{\text{zx}} \) is the sum of calls to CheckZero and Open, and \( \eta_{\text{cm}} \) the number of calls to CheckMult.

The proof of Theorem 5 is given in Appendix F.2. 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 adoption of the SPDZ-style sacrificing check from large fields to the \( \mathbb{Z}_{2^k} \) setting is secure.

**Lemma 4.** If \( P^* \) and \( V \) run the CheckMult protocol of \( \Pi_{\text{ComZK-b}} \) with inputs \( ([a_i], [b_i], [c_i])_{i=1}^n \) such that there exists an index \( 1 \leq i \leq n \) such that \( a_i \cdot b_i \neq k \cdot c_i \), then \( V \) outputs (success) with probability at most \( \eta_{\text{cm}}' + \epsilon_{\text{cm}} \) with \( \epsilon_{\text{cm}}' := 2^{-s} \), and \( \epsilon_{\text{cm}} \) the soundness error of CheckZero given in Lemma 2.

**Proof.** Suppose \( P^* \) and \( V \) run the CheckMult protocol with inputs as described in the lemma.

Since CheckZero' is a variant of CheckZero from \( \Pi_{\text{ComZK-b}} \), for the larger message space \( \mathbb{Z}_{2^k} \), we can apply Lemma 2 again: Hence, a \( P^* \) that tries to cheat during CheckZero' is detected by \( V \) except with probability \( \epsilon_{\text{cm}} \).

Now assume this does not happen, all the zero checks are correct, and \( V \) accepts. Let \( i \) be an index of an invalid triple such \( a_i \cdot b_i \neq k \cdot c_i \). Then, \( P \) has chosen \( z_i \in \mathbb{Z}_{2^k} \) such that

\[
0 \equiv_{k+s} \eta \cdot c_i - z_i - \epsilon_i \cdot b_i \equiv_{k+s} \eta \cdot c_i - z_i - \eta \cdot a_i \cdot b_i + x_i \cdot b_i
\]

\[
\iff
z_i - x_i \cdot b_i \equiv_{k+s} \eta \cdot (c_i - a_i \cdot b_i).
\]

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:

\[
(z_i - x_i \cdot b_i)/2^v \equiv_{k+s-v} \eta \cdot (c_i - a_i \cdot b_i)/2^v
\]
Much of the protocol is identical to $\Pi^Z_{\text{ComZK-a}}$ (Figures 14 and 15), with the exception that the MACs are now computed in the larger ring $\mathbb{Z}_{2k+2^s}$. \textbf{Init}, \textbf{Random}, \textbf{Affine Combination}, \textbf{Input} and \textbf{Open} work exactly as in $\Pi^Z_{\text{ComZK-a}}$, although using $F^{s,k+2s}_{\text{vole2k}}$.

\textbf{CheckZero} For $(\text{CheckZero}, [x_1], \ldots, [x_n])$, the parties proceed as follows:

1. If one of the $x_i$ is not equal to 0, then $\mathcal{P}$ aborts.
2. They run $[r] \leftarrow \text{Random()}$.
3. $\mathcal{V}$ samples $\chi_1, \ldots, \chi_n \in_R \mathbb{Z}_{2^s}$ 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 $m \equiv_{k+2s} 2^k \cdot \Delta \cdot p + \sum_{i=1}^n \chi_i \cdot K[x_i] + 2^k \cdot K[r]$, and outputs (success) if the equality holds and aborts otherwise.

\textbf{CheckZero’} $\text{CheckZero'}$ denotes a variant of the above which checks that $\tilde{x}_i = 0$ (mod $2^{k+s}$), 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 $\text{Open'}([x],[\ell])$ is similarly an adaption revealing the lower $k + s$ bits and using $\text{CheckZero'}$.

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

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

![Protocol $\Pi^Z_{\text{ComZK-b}}$ instantiating $F^Z_{\text{ComZK}}$ using a Mac’n’Cheese-style BMRS20 multiplication check.](image)

Since $(c_i - a_i \cdot b_i)/2^w$ is odd, it is invertible modulo $2^{k+s-v}$ and we can move it to the other side, getting
\[
(z_i - x_i \cdot b_i)/2^w \cdot ((c_i - a_i \cdot b_i)/2^w)^{-1} \equiv_{k+s-v} \eta.
\]
Since $k > v$, we have $k + s - v > s$, and the prover would have guessed all $s$ bits of $\eta \in \mathbb{Z}_{2^s}$ which happens only with probability $2^{-s}$. Therefore, by the union bound, $\mathcal{P}^*$ can make $\mathcal{V}$ output (success) with probability at most $\varepsilon_{ax} + 2^{-s}$.

5.3.3 Communication Costs

In the protocols $\mathcal{V}$ samples random coefficients $\chi_i$ in CheckZero and a permutation $\pi$ in CheckMult of $\Pi^Z_{\text{ComZK-a}}$ 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 $\lambda$ bits (for a computational security parameter $\lambda$) instead of $n \cdot s$ bits for CheckZero and $\log_2(n \cdot B + c)!$ bits for 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 $\Pi^Z_{\text{ComZK-a}}$.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>CheckZero</th>
<th>Open</th>
<th>CheckMult</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi^Z_{\text{ComZK-a}}$</td>
<td>0</td>
<td>$k$</td>
<td>$3Bk$ and $3B$ VOLEs</td>
</tr>
<tr>
<td>$\Pi^Z_{\text{ComZK-b}}$</td>
<td>0</td>
<td>$k$</td>
<td>$2 \cdot (k + s)$ and 2 VOLEs</td>
</tr>
</tbody>
</table>

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


A Proof of Security of the Basic Conversion Protocol

A.0.1 The RealGame

In order to prove the security of $\Pi_{\text{Conv}}$, we define an abstract game RealGame (Figure 17). In this abstraction, the prover (or $A$) 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 [1]). 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.

1. $A$ prepares $N + (NB + C)$ authenticated edaBits (representing the $N$ conversion tuples and $NB + C$ edaBits required to run the check), $\{([r_0^j], [r_0^j], [r_m^j]) \}_{j \in \{N + NB + C\}}$. $NB + C'$ potentially tampered circuits $\{C_j^*\}_{j \in \{NB + C\}}$ and lastly $NB$ daBits $\{(b_j^i, [b_j^i]) \}_{j \in \{NB\}}$. These are all sent 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 $([r_0], [r_m], [r])$ (where this is the first of the bucket) and $([s_0], [s_m], [s])$ take the next circuit $C^*$ and compute $(c_0, \ldots, c_m) \leftarrow C^*(r_0, \ldots, r, s_0, \ldots, s_m)$. Compute $c \leftarrow \sum_{i=0}^{m-1} c_i 2^i$ and check $r + s - 2^m c$.

$A$ wins if all of the checks pass and there is at 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 ($\bigcirc$) and bad (corrupt) edaBits are shades of gray balls (I.e. $\bullet$ or $\oplus$) where each shade defines a different kind of corruption. Likewise, a good circuit is a clear triangle ($\bigtriangleup$) and bad circuits are gray triangles ($\triangleleft$). 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. $A$ 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 $A$.

4. $A$ 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.

$A$ 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 $A$ 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 $[x_1, x_i|y_{i-1}] \in \{(\circ, \circ|\Delta), (\circ, \circ|\Delta), (\circ, \circ|\Delta)\}$ return reject.
   - If $[x_1, x_i|y_{i-1}] \in (\circ, \circ|\Delta)$ and $f(\circ, \circ|\Delta) = 0$ return reject.
   - If $[x_1, x_i|y_{i-1}] \in (\circ, \circ|\Delta)$ and $f(\circ, \circ|\Delta) = 0$ 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 \([B_1, B_2, B_3, B_4]\) and three triangles \([T_1, T_2, T_3]\). The buccetcheck procedure then checks all of the configurations \([B_1, B_2, T_1], [B_1, B_3, T_2], [B_1, B_4, T_3]\) and check if any of these configurations are \([\bullet, \bullet, \triangle], [\circ, \bullet, \triangle], [\bullet, \circ, \triangle]\) 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 \([\bullet, \bullet, \triangle]\) and in other cases it terminates due to \([\bullet, \bullet, \triangle]\). 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>
<thead>
<tr>
<th>Circles</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bullet, \bullet )</td>
<td>( \triangle )</td>
</tr>
<tr>
<td>( \circ, \bullet )</td>
<td>( \triangle )</td>
</tr>
<tr>
<td>( \bullet, \circ )</td>
<td>( \triangle )</td>
</tr>
<tr>
<td>( \bullet, \bullet )</td>
<td>( \triangle / \triangle )</td>
</tr>
<tr>
<td>( \circ, \bullet )</td>
<td>( \triangle / \triangle )</td>
</tr>
</tbody>
</table>

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

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 \( B \) that wins RealGame with non-negligible probability, then there exists an efficient adversary \( A \) against SimpleGame that wins with non-negligible probability. \( A \) will simulate the RealGame challenger and then use \( B \) to win SimpleGame.

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

As mentioned, the adversary \( A \) simulates the challenger of RealGame and then uses \( B \) to win SimpleGame. \( B \) sends a batch of edaBits and a set of circuits (and daBits) to \( A \). \( A \) randomly permutes the edaBits and then transforms them into circles. \( A \) 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.

\( A \) sends the set of circles to the challenger of SimpleGame, who then throws them randomly in buckets and sends these back to \( A \). The same happens with the set of triangles. In RealGame, \( A \) pairs the edaBits and the circuits according to the same arrangement as what was given to \( A \) from the SimpleGame challenger. These are then given to \( B \). If \( B \) is capable of winning RealGame, then \( B \) can be used to win the SimpleGame, as the configuration given by \( A \) to \( B \) 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 \( B \) wins RealGame with non-negligible probability, then \( A \) 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 \( N \geq 2^{s/B} \).

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

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

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 \( \alpha \geq 2^{s/B} - 1 \) and \( C = B \):

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

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

**Lemma 6.** The probability of A passing the second check in SimpleGame is less than \( 2^{-s} \), if the adversary corrupts more than \( \alpha \) fraction of triangles for \( \alpha = 2^{s/B} - 1 \) 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,

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

Using similar argumentation but bounding by a \( \beta \) fraction rather than \( \alpha \) and letting \( C' = C = B \), we conclude

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

**Lemma 7.** The probability of A passing the second check in SimpleGame is less than \( 2^{-s} \), if the adversary corrupts more than \( \beta \) fraction of triangles for \( \beta = 2^{s/B} - 1 \) 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 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.

\[
\{(\bullet, \bullet, \bullet), (\bullet, \bullet, \bullet), (\circ, \circ, \circ)\}
\]

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.

\[
\{(\Delta, \Delta, \Delta), (\Delta, \Delta, \Delta), (\Delta, \Delta, \Delta)\} \\
\{(\Delta, \Delta, \Delta), (\Delta, \Delta, \Delta), (\Delta, \Delta, \Delta)\}
\]

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 \((\bullet, \bullet, \Delta)\) or a bad triangle \((\bullet, \bullet, \Delta)\). In this example, let \( f(\bullet, \bullet, \Delta) = 1 \) (and \( f(\circ, \bullet, \Delta) = 1 \)) in which case the second arrangement is the favorable arrangement.

As a result of this discussion, the probability of passing 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 bucketcheck given a specific arrangement of balls \( L_i \) is given by

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

where \( t = NB\beta \). Thus,

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

as we know \( 0 \leq \beta \leq 1 \).

Now, to give an upper bound for \( \Pr[\mathcal{A} \text{ passes bucketcheck}] \) we provide an upper bound of the probability for different ranges of \( \alpha \) and \( \beta \) where the total probability is given by

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

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

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

We now try to bound \( \Pr[\mathcal{A} \text{ passes bucketcheck} | L_i] \). 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

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

This probability is maximal at \( t = B \) or \( t = NB - B \) as given by

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

\[
= \frac{B \cdot (B - 1) \cdots 1}{NB \cdot (NB - 1) \cdots NB - (B - 1)}
\]
Now, given that $N \geq \frac{2^s}{B}$, we arrive at

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

Lastly we note that

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

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

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

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

Case II. Let $t > NB - B$. Whenever $t$ is greater than $NB - B$, $A$ 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 $A$ 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

$$\{(\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot)\}$$

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 $A$ 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 $A$ to actually cheat in 3 buckets rather than 2, since if $A$ hits the following arrangement of triangles

$$\{(\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot), (\cdot, \cdot, \cdot)\}$$

the bucket check could pass with three conversion tuples instead. To remedy this situation, observe that $A$ 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 $A$ 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 $A$ 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 $A$. 

30
\[
\begin{align*}
\{(\bullet, \bullet, \bullet), (\bullet, \bullet, \bullet), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \bullet, \bullet), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\end{align*}
\]

Note that the darker balls $\bullet$ are such that $f(\bullet, \bullet, A) = 1$ in all the listed configurations, forcing $A$ to specifically hit the $\Delta$ 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 $A$ in the bucket check can be expressed as

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

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 $\bullet$ (represented as $\circ$) 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 $A$ 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 $K = N - 1$.

\[
\begin{align*}
\binom{N}{N-1} \left( \frac{3(N-1)}{1} \right) \left( \frac{3(N-(N-1))}{1} \right) \frac{1}{\binom{3N}{3(N-1)+1}} \frac{1}{\binom{3N}{2}} \\
= N \cdot (3N - 3) \cdot 3 \cdot \frac{1}{\binom{3N}{2}} \cdot \frac{1}{\binom{3N}{2}} \\
= \frac{3N - 3}{3N - 1} \cdot \frac{2}{3N - 1} \cdot \frac{2}{3N - 1} \\
\leq \frac{3N - 3}{3N - 1} \cdot 2^{-s/2} \cdot 2^{-s/2} \leq 2^{-s}, \text{ given } N \geq 2^{s/2}
\end{align*}
\]

**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 $A$ when $K$ is fixed. For example, for $N = 4$ and $K = 2$, there are the 10 cases:

\[
\begin{align*}
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\{(\bullet, \bullet, \bullet), (\bullet, \circ, \circ), (\circ, \circ, \circ), (\circ, \circ, \circ)\} \\
\end{align*}
\]

31
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 $A$. 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,

$$Pr[A \text{ passes bucketcheck}] \leq \left( \frac{N}{N-1} \right) \left( \frac{(N-1) \cdot 4}{2} \right) \cdot \left( \frac{4}{1} \right) \cdot \frac{1}{4(N-1)+1} \cdot \frac{1}{(4N)^3}$$

$$= N \cdot \left( \frac{4N-4}{2} \right) \cdot 4 \cdot \frac{(4N-3) \cdot 3!}{(4N)!} \cdot \frac{3! \cdot (4N-3)!}{(4N)!}$$

$$= 8(N-1) \cdot (4N-5) \cdot \frac{3}{4N} \cdot \frac{3}{4N} \cdot \frac{2}{4N-1} \cdot \frac{2}{4N-1} \cdot \frac{1}{4N-2} \cdot \frac{1}{4N-2}$$

$$\leq 8(N-1) \cdot (4N-5) \cdot (2^{-s/3})^6, \text{ given } N \geq 2^{s/3}$$

**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 $A$ when $K$ is a fixed integer. For instance, for $N = 4$ and $K = 2$, these are the cases.

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

$$\{(\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet), (\bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet, \bullet \bullet \bullet \bullet)\}$$

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.

$$Pr[A \text{ passes bucketcheck}] \leq \left( \frac{N}{N-1} \right) \left( \frac{(N-1) \cdot 5}{2} \right) \cdot \left( \frac{5}{2} \right) \cdot \frac{1}{(5N-1)+1} \cdot \frac{1}{(5N)^4}$$

$$= N \cdot \left( \frac{5N-5}{2} \right) \cdot \left( \frac{5}{2} \right) \cdot \frac{(5N-3) \cdot 3!}{(5N)!} \cdot \frac{4! \cdot (5N-4)!}{(5N)!} \leq 2^{-s}, \text{ given } N \geq 2^{s/4}$$

We summarize the analysis as follows.
Lemma 8. The probability of 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 $\Pi_{\text{Conv}}$ (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 $S$ given access to $\mathcal{F}_{\text{Conv}}$ that runs the adversary $A$ as a subroutine. We implicitly assume that $S$ passes all communication between $A$ and $Z$.

**Malicious Prover.** $S$ sends $(\text{corrupted}, P)$ to the ideal functionality $\mathcal{F}_{\text{Conv}}$. $S$ creates a copy of the verifier $V$, and runs this verifier according to the protocol $\Pi_{\text{Conv}}$, while letting the prover $P^*$ behave as instructed by the adversary $A$.

1. In the setup-phase, $P^*$ 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 $\{(x', y', z')\}_{j \in [NBm+Cm]}$. $S$ records and forwards all of these to $\mathcal{F}_{\text{Conv}}$.
2. $S$ runs $\mathcal{F}_{\text{Dabits}}$ with the input provided by $P^*$. If $\mathcal{F}_{\text{Dabits}}$ returns abort, then send abort to $\mathcal{F}_{\text{Conv}}$ and terminate. Otherwise continue.
3. $S$ 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 $P^*$ before shuffling the values provided by $P^*$.
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 $\mathcal{F}_{\text{Conv}}$ and terminate.
5. For each bucket during the Choose-phase, $S$ emulate bitADDcarry by running like an honest verifier and convertBit2A by calling this using $\mathcal{F}_{\text{Conv}}$.
6. $S$ runs the rest of the protocol as an honest verifier. If the honest verifier outputs abort, then $S$ sends abort to $\mathcal{F}_{\text{Conv}}$ and terminate. If the honest verifier outputs (success), then $S$ sends $(\text{VerifyConv}, N, \{(c^j)_{[2], \ldots, [m]}[j]_{[M]}\}_{j \in [N]})$ to $\mathcal{F}_{\text{Conv}}$ (essentially forwarding the call made originally by $P^*$).

The messages that $P^*$ receives from $S$ have the same distribution as in the real protocols. Whenever the verifier simulated by $S$ outputs abort (as in the protocol), then the verifier in the ideal setting outputs abort as well (since $S$ sends abort to $\mathcal{F}_{\text{Conv}}$). The only case where $P^*$ may distinguish between the ideal and real, is if the simulated verifier run by $S$ outputs (success) but at least one conversion tuple is inconsistent, in which case $\mathcal{F}_{\text{Conv}}$ will abort. But if at least one conversion is inconsistent, then by Theorem 1 the probability with which $P^*$ avoids being caught in Step 3 of $\Pi_{\text{Conv}}$ is at most $2^{-s}$.

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

1. $S$ samples random values corresponding to the edaBits and daBits $\{(r_0^j, \ldots, r_{m-1}^j, r^j)\}_{j \in [NB+C]}$, daBits $\{(b^j, b'^j)\}_{j \in [NB+s]}$ and triples $\{(x', y', z')\}_{j \in [NBm+Cm]}$ and commits to these by calling $(\text{Input}, \cdot, \cdot)$ using $\mathcal{F}_{\text{ComZK}}^{2M}$ in the appropriate domain.
2. If $V^*$ at any outputs abort, then send abort to $\mathcal{F}_{\text{Conv}}$ and terminate.
3. $S$ sends $(\text{VerifyDabit}, NB + s, \{(b_j)_2, (b_j)_{2M}\})_{j \in [NB+s]}$. If this call returns abort, then output abort and terminate. Otherwise continue.

4. On receiving $\pi_1, \pi_2, \pi_3$, $S$ locally shuffles the sampled values.

5. During the Cut-phase, $S$ opens the last $C$ edaBits, triples and daBits honestly, as it knows the underlying values.

6. During the remainder of the protocol, $S$ runs like an honest prover.

7. Lastly, $S$ sends $(\text{VerifyConv}, N, \{(c_0)_{2j}, \ldots, (c_{m-1})_{2j}, (c_j)_M\})_{j \in [N]}$ to $F_{\text{Conv}}$ (again, forwarding the initial call) and outputs whatever $V$ outputs.

In both the ideal and real execution all values sent from the honest prover (or simulator) to the verifier, are hidden by $F_{\text{ComZK}}^{2M}$. 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 $F_{\text{ComZK}}^{2M}$. Any calls to VerifyDabit will fail in the real only if the same call would fail in the ideal world, due to the usage of $F_{\text{FDabit}}$ in both. Same goes for CheckZero. Lastly, 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 $V^*$ simulated by $S$ is distributed identically to its view in the real protocol execution.

\section*{B Proof of Conversion Protocol for Faulty Dabits}

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

This functionality extends $F_{\text{ComZK}}^{2M}$ with the extra function VerifyDabit that takes a set of IDs $(id^{0,j}, id^{1,j})_{j \in [N]}$ of (potentially faulty) daBits $(id^{0,j}, id^{1,j})_{j \in [N]}$ and verifies that $b^{0,j} \equiv b^{1,j} \mod 2$ where $b^{0,j} \in \mathbb{Z}_2$ and $b^{1,j} \in \mathbb{Z}_M$ 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. $F_{\text{FDabit}}$ communicates with two parties $P, V$.

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

- If $b^{0,j} \equiv b^{1,j} \mod 2$ for all $j \in [N]$, then output (success) to $V$, 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

$$\Pr\left[\left|\sum_{i=1}^N e_i b_i \mod p\right| < 1/2 \cdot \max\{|b_i|\} \leq 1/2\right].$$
A proof for this can e.g. be found in [BL17, Lemma 2.3].

**Proof of Lemma 4 (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 $F_{FDabit}$ and emulating $F^{2,M}_{\text{Com2K}}$ creates a correct distribution of transcripts.

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

1. $S$ gives the provided set of id’s to $F_{FDabit}$, given what $P^*$ inputs into $F^{2,M}_{\text{Com2K}}$.

2. For each of the $s$ iterations, $S$ follows the protocol honestly.

3. If the protocol succeeds, then it runs $\text{VerifyDabit}$ in $F_{FDabit}$, otherwise it sends abort.

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

Consider the sum $r' = \sum_{i=1}^{N} c_i b_i$. 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 $r = \tau = e_0 + \sum_{i=1}^{N} e_i b_i \mod 2$. Now, except with probability $2^{-s}$, the claim on the parity of the $\text{daBits}$ follows by a standard argument.

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

1. $S$ sends $(\text{VerifyDabit}, N, \{b^j\}_2, \{b^*\}_M)$ to $F_{FDabit}$. If $F_{FDabit}$ returns abort, then $S$ outputs abort to $V^*$ and terminate. Otherwise continue.

2. Since $F_{FDabit}$ did not abort, we know that it holds for each $j \in [N]$ that $b^j \equiv b^* \mod 2$. The simulator executes the remainder of the protocol $\Pi_{FDabit}$ as follows:

   - For $r$ it opens a uniformly random bit.
   - Then it reveals a uniformly random $\tau \in [0, 2^\gamma - 1]$ such that $\tau = r \mod 2$.

The view of $V^*$ simulated by $S$ is distributed equally to its view in the real world. We consider what the corrupted verifier $V^*$ sees during an execution of $\Pi_{FDabit}$ 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 $\text{bitADDcarry}$ procedure. The purpose of this discussion is to distinguish between the potential errors caused by a faulty $\text{daBit}$ 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 bitADDcarry (as described in Figure 2 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 $\Pi_{\text{Conv}}$), the error essentially causes the adding or subtraction of an arbitrary value $\text{err}_{\text{triples}} < 2^m$ (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 $i = 0, \ldots, m - 1$ and/or the carry bit $[e_m]_2$ is incorrect. This error can then be used to force equality of the check $e = \sum_{i=0}^{m-1} 2^i \cdot e_i$ (for $e = [r + s]_M - 2^m \cdot [e_m]_M$) in the case of an inconsistent conversion tuple, by masking the error $\text{err}_{\text{triples}} < 2^m$, 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 bitADDcarry evaluation will be bits, although they may lead to arbitrary error $0 \leq \text{err}_{\text{triples}} < 2^m$.

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

In 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 $\Pi_{\text{Conv}}$ in Section 3.3. Consider the case where the carry bit of bitADDcarry is tampered such that it would have been $x = 0$, but was flipped to $x_{\text{err}} = 1$. When $x_{\text{err}}$ is then input to 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 convertBit2A. This provides an adversarial prover more options in winning SimpleGame and as much must be disallowed.

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

This $x_{\text{err}}$ (authenticated in $\mathbb{Z}_M$) is then used in the computation $[e']_M = [r + s]_M - 2^m \cdot x_{\text{err}}$ 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 $e' = \sum_{i=0}^{m-1} 2^i \cdot c_i$ (where $c_i, i = 0, \ldots, m - 1$ are the bits from the sub-protocol bitADDcarry). Now, if $x_{\text{err}}$ 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 $2^m \cdot x_{\text{err}}$ being much larger than anything which can be represented in $m$ bits, given that $x_{\text{err}} \notin \{0, 1\}$. To this end, we conclude that a faulty daBit such that $x_{\text{err}} \notin \{0, 1\}$ will allow an adversary to pass $\Pi_{\text{Conv}}$ using an inconsistent conversion tuple $([r_0], \ldots, [r_{m-1}], [r]_M)$ with an error of $\text{err}_{\text{daBit}} = r - \sum_{i=0}^{m-1} r_i \geq 2^m$.

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 \( (r - \sum_{i=0}^{m-1} r_i) = \text{err} \notin \{0, 1\} \) designed to pass \( \Pi_{\text{Conv}} \) using faulty multiplication triples, it must hold that \( \text{err} < 2^m \), while if the conversion tuple was designed to pass using faulty dabits, \( \text{err} \geq 2^m \).

### B.1 Security Analysis of the Faulty Dabit Approach

To accommodate for the daBits used within the conversion verification, we now modify \textbf{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 \textbf{RealGame}_{\text{faulty}} (Figure 21) to that of our old \textbf{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 \textbf{RealGame}_{\text{faulty}} in terms of the \textbf{SimpleGame} syntax from the original proof. We formally define the revised \textbf{RealGame}_{\text{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 \textbf{RealGame}_{\text{faulty}} differs from the \textbf{RealGame} defined in Figure 17.

#### The \textbf{RealGame}_{\text{faulty}} with faulty daBits

1. \( A \) prepares must now prepare \( NB + C \) daBits \( \{(\langle j \rangle_2, \langle j \rangle_M)\}_{j \in \{NB+C\}} \).
2. The challenger opens \( C \) edaBits, circuits, and daBits now. If any of the edaBits, daBits or circuits are inconsistent, \textbf{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 \textbf{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 \( (c_0, \ldots, c_m) = C'(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1}, b_2, b_M) \). 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 \textbf{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 \( c \geq r + s - 2^m \cdot c'_m \).

Figure 21: The revised \textbf{RealGame}_{\text{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 \textbf{RealGame}_{\text{faulty}}.

#### Lemma 10. Security against all adversaries of the \textbf{SimpleGame} implies security against all adversaries of the \textbf{RealGame}_{\text{faulty}}.

**Proof.** The only difference between this reduction and the original (Lemma 5), is that \( A \) now pairs up circuits and daBits and converts these into triangles, rather than only the circuits.
As the inconsistency $\err_{\text{daBit}}$ in conversion tuples that may pass $\Pi_{\text{Conv}}$ using a faulty $\text{daBit}$ differs enough from the inconsistency $\err_{\text{triple}}$ allowed by faulty multiplication triples (specifically $\err_{\text{triple}} < \err_{\text{daBit}}$), we conclude that we can treat them as one entity comparative to the tamper-resilient circuits of the original $\text{RealGame}$, simply allowing for a wider range of errors. This means the above proof sketch is a sufficient reduction from the $\text{RealGame}_{\text{faulty}}$ to the $\text{SimpleGame}$, allowing us to re-use our proof from Lemma 5.

C Proof of Correct Truncation

**Theorem 3.** The protocol $\Pi_{\text{VerifyTrunc}}$ (Figure 13) UC-realizes $F_{\text{VerifyTrunc}}$ (Figure 9) in the $F_{\text{CheckLength}}$-hybrid model.

Before writing the proof, we make the following observations. First, if correct information is provided by $P$, 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 $\text{CheckZero}$.

- **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 $S$ given access to $F_{\text{VerifyTrunc}}$ that will emulate $F_{\text{CheckLength}}$. We implicitly assume that $S$ passes all communication between the adversary (either $P^*$ or $V^*$ dependent on the case) and the environment $Z$.

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

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

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

1. $S$ forwards the call $(\text{VerifyTrunc}, N, \{m^j, [a']_M, [a_{tr}]_M\}_{j \in [N]})$. If $F_{\text{VerifyTrunc}}$ returns $\perp$, output $\perp$ to $V^*$ and abort.
2. For each $j \in [N]$ $S$ commits to a random value $[a']_M$ using Input of $F_{\text{CheckLength}}$. We assume that simulated commitments to $a', a^*$ already exist in $F_{\text{CheckLength}}$. 38
3. For each iteration \( j \in [N] \), let \( l \) be the size of \( a^j \) and \( m \) be the size of \( a^\prime j \). \( S \) runs \((\text{CheckLength}, \text{id}^{a^\prime j}, m)\) and 
\((\text{CheckLength}, \text{id}^{a^\prime j}, l - m)\) in \( F_{\text{CheckLength}} \) towards the verifier.

4. \( S \) then computes \( y^j \leftarrow a^j - (2^m \cdot a^\prime j + a^\prime j) \) using \( F_{\text{CheckLength}} \) and then runs \((\text{CheckZero}, \text{id}^{y^j})\), which it makes output \( \text{success} \).

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

\[ \square \]

## D A Naïve Truncation Protocol

For comparison, we now describe a “naïve” way of truncating some value \([a]_M\) where \( a \in [0, 2^3] \subset \mathbb{Z}_M \), 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 \([a_{tr}]_M\).

\[
\text{Protocol} \ \Pi_{\text{NaiveTrunc}}
\]

**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 \( \text{CheckZero}([y]_M, [y_0]_M, \ldots, [y_{l-1}]_M) \), output \text{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 naively truncates \( a \) by \( m \) bits.

This protocol is much more expensive than our \texttt{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 \texttt{CheckZero}. All calls to \texttt{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 \( a_i (1 - a_i) \neq 0 \) 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 $\pi_{\text{NaiveTrunc}}$ (Figure 22) and $\pi_{\text{VerifyTrunc}}$ (Figure 12).

<table>
<thead>
<tr>
<th></th>
<th>#Openings $F_2$</th>
<th>#Openings $Z_M$</th>
<th>#Faulty triples $F_2$</th>
<th>#Faulty triples $Z_M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve log($M$) $\leq s$</td>
<td>0</td>
<td>$l((B - 1) \cdot 2) + 2$</td>
<td>0</td>
<td>$(B - 1)l$</td>
</tr>
<tr>
<td>Naïve log($M$) $&gt; s$</td>
<td>0</td>
<td>$l \cdot 2 + 2$</td>
<td>0</td>
<td>$l$</td>
</tr>
<tr>
<td>Ours</td>
<td>$Bl + 2B$</td>
<td>$2B + 1$</td>
<td>$Bl$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>#(e)daBit COTs</th>
<th>#(e)daBit VOLEs</th>
<th>#Bits from fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve log($M$) $\leq s$</td>
<td>0</td>
<td>0</td>
<td>$2(B - 1)l \log_2(M)$</td>
</tr>
<tr>
<td>Naïve log($M$) $&gt; s$</td>
<td>0</td>
<td>0</td>
<td>$2l \cdot \log_2(M)$</td>
</tr>
<tr>
<td>Ours</td>
<td>$Bl + 2B$</td>
<td>$4B$</td>
<td>$(B + 1)l + (4B + 2) \log_2(M)$</td>
</tr>
</tbody>
</table>

A breakdown of the costs of $\pi_{\text{NaiveTrunc}}$ compared to those of our optimized protocol $\pi_{\text{VerifyTrunc}}$ (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 $Z_M$ openings scales with the bit-length $l$. To give a concrete number, e.g. for the $Z_p$ 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 $Z_2$ and $Z_p$ for both protocols.

E Sub-protocols

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

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \land (y_i \oplus c_i)), \forall i \in \{0, \ldots, m - 1\}$$

(3)

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

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

(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 $F_2$ to $F_M$. 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. $c \leftarrow \text{Open}([c][2])$
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. \(\text{CDE}^{+18}\) which we cite here:

**Lemma 11** (Lemma 1 from \(\text{CDE}^{+18}\)). Let \(\ell, r \) and \(m\) be positive integers such that \(\ell - r \leq m\). Let \(\delta_0, \delta_1, \ldots, \delta_t \in \mathbb{Z}\), and suppose that not all the \(\delta_i\)'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 \(\alpha\) below, we have

\[
\Pr_{\alpha, x_1, \ldots, x_n \sim \mathcal{R}_{\mathbb{Z}_{2^m}}}[\alpha \cdot (\delta_0 + \sum_{i=1}^{t} \chi_i \cdot \delta_i) \equiv \ell \pmod{y}] \leq 2^{-\ell + r + \log(\ell - r + 1)}.
\]

**Proof of Lemma 2**

This proof is similar to the proof of Claim 1 in \(\text{CDE}^{+18}\).

Since \((x_1, \ldots, x_n) \not\equiv_k (0, \ldots, 0)\), there is at least one \(i\) with \(x_i \not\equiv_k 0\). Let \(p' = p + \varepsilon \in \mathbb{Z}_{2^k}\) 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 \(V\) to accept, \(m'\) must satisfy the following equality:

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

Subtracting (the honestly computed)

\[
m = \sum_{i=1}^{n} \chi_i \cdot M[x_i] + 2^k \cdot M[r]
\]

we get

\[
m = \sum_{i=1}^{n} \chi_i \cdot (\Delta \cdot x_i + K[x_i]) + 2^k \cdot (\Delta \cdot r + K[r])
\]

\[
= 2^k \cdot \Delta \cdot p + \Delta \cdot \sum_{i=1}^{n} \chi_i \cdot x_i + \sum_{i=1}^{n} \chi_i \cdot K[x_i] + 2^k \cdot K[r]
\]

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 \(\text{CDE}^{+18}\) (repeated as Lemma 11 on p. 41) with the following variables

\[
\alpha := \Delta \quad \delta_0 := 2^k \cdot \varepsilon \quad \delta_i := -x_i \quad \chi_i := \chi_i \\
\ell := k + s \quad m := s \quad r := k
\]

and get \(2^{-\ell + r + \log(\ell - 1 + 1)} = 2^{-s + \log(s + 1)}\) as upper bound on the success probability of \(P^*\) finding \(m' - m\), which is necessary to make \(V\) output (success).
F.2 Proof of Lemma 3

Proof of Lemma 3 Suppose \( P^* \) and \( V \) run the 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 [Bec92]. So the verifier outputs (failure).

Therefore, \( P^* \) has two possible options: 1. It can try to cheat during the CheckZero in Step 7 to reveal some different values \( d', e' \neq d, e \) or \( w_k \neq 0 \) in Step 8. This succeeds with probability at most \( \varepsilon_{\text{cz}} \) (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 \( \varepsilon_{\text{cm}} = (\frac{n\cdot B\cdot C}{B})^{-1} \).

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

F.3 Proof of Theorem 4

Proof of Theorem 4 To show security in the UC-model, we construct a simulator \( S \) with access to the ideal functionality \( F_{\text{ComZK}}^{Z_{2^k}} \). The environment can choose to corrupt one of the parties, whereupon \( S \) 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 \( S \) sends (corrupted, \( P \)) to the ideal functionality \( F_{\text{ComZK}}^{Z_{2^k}} \). It also creates copies of the prover \( P^* \) and verifier \( V \), and runs the verifier according to the protocol \( \Pi_{\text{ComZK}, A}^{Z_{2^k}} \), while letting the prover behave as instructed by the environment. For this, \( S \) simulates the functionality of \( F_{\text{vole2k}}^{s, k + s} \) with corrupted \( P \). If the simulated \( P \) aborts the protocol, \( S \) sends (abort) to \( F_{\text{ComZK}}^{Z_{2^k}} \). The method calls are simulated as follows:

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

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

For CheckZero, \( S \) first simulates the call to Random, and runs the protocol with the simulated parties. Then it sends the CheckZero message to \( F_{\text{ComZK}}^{Z_{2^k}} \). If the simulated verifier aborts, then \( S \) sends (abort) to \( F_{\text{ComZK}}^{Z_{2^k}} \), 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 (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 \( F_{\text{ComZK}}^{Z_{2^k}} \), the premise hold if one of the input commitments contains a non-zero value. Thus, we can apply Lemma 3 which gives us the desired consequence.

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

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

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

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

**Malicious Verifier** The setup of the simulation in case of a corrupted verifier $V^*$ is similar as before. $S$ sends (corrupted, $V$) to the ideal functionality $F_{\text{ComZK}}^{x,k}$. It creates copies of the prover $P$ and verifier $V^*$. The prover is run according to the protocol, whereas the environment controls the verifier. For this, $S$ simulates the functionality of $F_{\text{vole2k}}^{x,k+s}$ with corrupted $V$. For all methods, since $V$ 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 $F_{\text{ComZK}}^{x,k}$. The method calls are simulated as follows:

During initialization, $S$ allows $V^*$ to choose its MAC key $\Delta$ with the simulated $F_{\text{vole2k}}^{x,k+s}$ functionality.

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

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

For CheckZero, $S$ sends the respective message to $F_{\text{ComZK}}^{x,k}$. If it aborts, then $S$ instructs the simulated $P$ to also abort by sending (abort) to the simulated $V$, which finishes the simulation. Otherwise, $S$ simulates the normal protocol execution: It first simulates the call to Random and lets $V^*$ choose the coefficients $\chi_i$. Since $F_{\text{ComZK}}^{x,k}$ 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 $p \in_R Z_2$, and compute $m := 2^i \cdot \Delta \cdot p + \sum_{i=1}^n \chi_i \cdot K[x_i] + 2^k \cdot K[r]$ as expected by the verifier.

For Input, $S$ first simulates the call to Random as above, and then sends a random value $\delta \in_R Z_2$ to the simulated verifier. Also, $S$ sends (Input) on behalf of $V^*$ to $F_{\text{ComZK}}^{x,k}$.

For Open, $S$ sends the Open on behalf of $V^*$ to $F_{\text{ComZK}}^{x,k}$ and receives the committed values $x_1, \ldots, x_n \in Z_2$ as output. It sends these values to the simulated verifier, and then simulates Affine and CheckZero as above. So the view is distributed identically to the real protocol.

For CheckMult, $S$ sends the corresponding message on behalf of the corrupted verifier to $F_{\text{ComZK}}^{x,k}$. If it aborts, then $S$ instructs the simulated $P$ to also abort by sending (abort) to the simulated $V$. Otherwise, $S$ simulates the complete protocol using the constant value 0 for all of the prover’s commitments. Because the simulated $P$ behaves like an honest prover, it samples all multiplication triples $(x, y, z)$ uniformly distributed in $Z_2$. Since the view of the $V$ is distributed identically to the real execution and independent of the prover’s real inputs: The opened triples in Step 8a are uniformly distributed, valid multiplication triples. The values $d, e$ revealed in Step 8a are distributed uniformly in $Z_2$, and the 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.

\[\Box\]

### F.4 Proof of Theorem 5

**Proof of Theorem 5** Since most of $\Pi_{\text{ComZK},b}^{x,k}$ is actually identical to $\Pi_{\text{ComZK},a}^{x,k}$, we will refer to the Proof of Theorem 4 for these parts, and focus on the differences here.
The subroutines CheckZero and CheckZero’ are only very slightly modified from the CheckZero from Π_{ComZK}^{Z_2^k}. The latter is exactly the same as in before, but for the larger message space $Z_{2^{k+s}}$, 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 CheckMult:

**Malicious Prover**  The setup of the simulation is the same as in the Proof of Theorem 4, i.e. $S$ sends (corrupted, $P$) to the ideal functionality $F_{\text{ComZK}}^{Z_2^k}$ and simulates copies of prover and verifier.

For the method CheckMult, $S$ can exactly simulate the protocol since it knows all the commitments, and $\eta$ is sampled uniformly at random from $Z_{2^s}$.

If the simulated verifier aborts, it sends (abort) to $F_{\text{ComZK}}^{Z_2^k}$. 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 $\varepsilon_{cz} + \varepsilon_{cm}'$.

**Malicious Verifier**  Again, we have the same setup as before, i.e. the simulator sends (corrupted, $V$) to the ideal functionality $F_{\text{ComZK}}^{Z_2^k}$ and simulates copies of prover and verifier.

For CheckMult, we use the same strategy as in the Proof of Theorem 4: $S$ sends the corresponding message on behalf of the corrupted verifier to $F_{\text{ComZK}}^{Z_2^k}$. If it aborts, then $S$ instructs the simulated $P$ to also abort by sending (abort) to the simulated $V$. Otherwise $S$ 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.