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

Carsten Baum
Aarhus University
Aarhus, Denmark
cbaum@cs.au.dk

Lennart Braun
Aarhus University
Aarhus, Denmark
braun@cs.au.dk

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

Benoit Razet
Galois, Inc.
Portland, Oregon, United States
benoit.razet@galois.com

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

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.

CCS CONCEPTS

• **Security and privacy** → *Cryptography*; • **Theory of computation** → *Cryptographic protocols.*

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484812

KEYWORDS

Zero-Knowledge Protocols; Commit and Prove; Rings; Conversion

ACM Reference Format:
Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. 2021. Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and $\mathbb{Z}_{2^k}$. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3460120.3484812

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 [18, 21, 25, 35] and vector oblivious linear evaluation (VOLE) [4, 15, 31, 33]. 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 [7, 8, 30, 34] from the LPN assumption. This has been the approach taken in recent VOLE-based ZK protocols [4, 31], which exploit the high computational efficiency and low communication overhead of LPN-based VOLE.

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

In the MPC setting, Escudero *et al.* [16] 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`, [16] 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 [16] starts by first creating private `edaBits` known to one party, and then summing these up across the parties to obtain secret-shared `edaBits`. In the ZK setting, the prover knows the values of the `edaBits`, so the second phase can clearly be omitted.

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

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

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

In our protocol, we introduce several optimizations of this approach, tailored to the ZK setting. Firstly, we observe that in the ZK setting, it is not necessary to create *random* verified `edaBits`, if we can instead just apply the `edaBit` verification protocol to the actual conversion tuples $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$, from the witness. This change would remove the need for the binary addition circuits in the last step. Unfortunately, the protocol of [16] 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 `edaBits` $((\![x]\!]_2, [x]_M)$ of length $m = 1$. They are used in the consistency check of [16] 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 [16], which also used `edaBits` for these operations. However, a drawback of these protocols is that in addition to `edaBits`, they use auxiliary binary comparison circuits, which add further costs. We show that in the ZK setting, these can be avoided, and obtain protocols which only rely on our efficient conversion check.

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

ZK for Arithmetic Circuits over $\mathbb{Z}_{2^k}$. Our conversion, truncation and comparison protocols can all be made to work with either a field $\mathbb{Z}_p$, or a ring $\mathbb{Z}_{2^k}$, giving flexibility in high-level applications. While ZK protocols for $\mathbb{Z}_p$ and $\mathbb{Z}_2$ have been well-studied, there is less work on protocols for circuits over $\mathbb{Z}_{2^k}$, especially in the commit-and-prove setting. We take the first step towards this, by showing how to use VOLE-based information-theoretic MACs for ZK over $\mathbb{Z}_{2^k}$, by adapting the techniques from SPD$\mathbb{Z}_{2^k}$ [12]. Given the MACs, which serve as homomorphic commitments in $\mathbb{Z}_{2^k}$, we show how to efficiently verify multiplications on committed values. We present two possible approaches: the first is based on a simple cut-and-choose procedure, adapted from [31] for binary circuits;

in the second approach, we adapt the field-based multiplication check from [4] 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 [7]). Removing the trusted setup can be done with an actively secure VOLE protocol over $\mathbb{Z}_{2^k}$. We note that the LPN-based construction of [7] also works over $\mathbb{Z}_{2^k}$ (as implemented in [30]), although currently only with passive security. It is an interesting future direction to extend efficient actively secure protocols [8, 31] to the $\mathbb{Z}_{2^k}$ setting.

Concrete Efficiency. We analyze the efficiency of our protocols, in terms of the bandwidth requirements and the amount of VOLE or OT preprocessing that is needed. We moreover present benchmarks based on an implementation of our conversion protocol and estimate the cost of $\mathbb{Z}_{2^k}$-VOLEs that are necessary for our $\mathbb{Z}_{2^k}$-protocols.

Compared to a "baseline" protocol consisting of a straightforward application of `edaBits` [16] to ZK, our optimized conversion protocol reduces communication by more than 2×, while also reducing the number of used VOLEs by around 4×. When comparing to a "naive" solution that decomposes the input as bits modulo p, we reduce both the overall communication and the required number of VOLEs by a factor m where m is the bit-length of the value.

In our implementation, we show the concrete efficiency of our conversion protocols. For example, to convert 2^{10} 32-bit values our system requires 9.6 s using [31] (9.3 ms amortized per conversion) and 7.5 s using [33] (7.3 ms amortized). For 2^{20} 32-bit elements, this increases to 181.7 s for [31] (173 µs amortized) and 92.1 s for [33] (87.8 µs amortized).

Our $\mathbb{Z}_{2^k}$ Zero-Knowledge proofs achieve amortized communication costs of $k + s$ bits and consume one VOLE to open a commitment (where s is a statistical security parameter and $k + s$ bits are necessary to represent committed ring element), and $2k + 4s$ bits plus three VOLEs, to verify a multiplication. This is competitive with state-of-the-art protocols for large fields such as [4, 31], which need to transfer 2–3 field elements .

1.2 Related Work

Our work builds upon concretely efficient zero-knowledge protocols from VOLE, which were first given in [4, 15, 31]. While [4, 31] use VOLE-based information-theoretic MACs in a black-box way, Line-Point ZK [15] takes a non-black-box approach, which reduces communication to just 1 field element per multiplication in a large field. More recently, Quicksilver [33] extends this to arbitrary fields, including boolean circuits. Since the core of our protocol uses potentially faulty information to verify `edaBits`, the techniques from QuickSilver could be plugged in to cheaply verify these faulty components, which would simplify much of our security analysis and slightly reduce costs. We analyze this approach in Section 6. This is very similar to the approach taken in the concurrent work Mystique [32], which uses Quicksilver directly for conversions. Since QuickSilver and Mystique make non-black-box use of VOLE-based MACs, these would not be applicable in settings based on other types of homomorphic commitments, or applications such as proofs of disjunctions in [4], which assumes a black-box commitment scheme. Thus, while our protocol has higher communication costs, it is more general and may be of use in a wider range of applications.

Another related work is Rabbit [26], 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 [10] 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 [10] relies on large groups.

2 PRELIMINARIES

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

2.1 Notation

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

2.2 Zero-Knowledge Proofs

Zero-knowledge proofs (of knowledge) are interactive two-party protocols that allow the prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a certain statement is true (and that it possesses a witness to this fact). This happens in a way such that $\mathcal{V}$ does not learn anything else besides this fact that it could not compute by itself. Instead of using the classical definition by Goldwasser et al. [19], we define zero-knowledge using an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ for satisfiability of circuits C: On input (Prove, C, w) from $\mathcal{P}$ and (Prove, C) from $\mathcal{V}$ the functionality $\mathcal{F}_{\mathsf{ZK}}$ outputs $\top$ to $\mathcal{V}$ iff. $C(w) = 1$ holds, and sends $\perp$ otherwise [31].

Following previous works (e.g. [4, 31]), we use the commit-and-prove strategy to instantiate $\mathcal{F}_{\mathsf{ZK}}$ using homomorphic commitments (see Section 2.4). These allow the prover $\mathcal{P}$ to commit to its witness w. Then the circuit 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) [17] 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) [2] 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 [22], OT extension [23] 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) [24, 27] is an arithmetic generalization of COT allowing a receiver to evaluate a secret linear equation $\alpha \cdot X + \beta$ (over a field $\mathbb{F}_p$ or ring $\mathbb{Z}_{2^k}$) held by the

sender at a point of its choice x to obtain $y = \alpha \cdot x + \beta$. This can be extended into *vector* OLE (VOLE) [1] where x and β are vectors of the same length rather than single field elements. *Subfield* VOLEs [9] extends this concept such that the elements of α and β live in an extension field $\mathbb{F}_{p^r} \supset \mathbb{F}_p$. Random (subfield) VOLE, where 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 [12] to the zero-knowledge setting. For fields $\mathbb{Z}_p$, we use BeDOZa-style MACs [6] which can be generated as follows: To authenticate values $x_1, \ldots, x_n \in \mathbb{Z}_p$ known to $\mathcal{P}$, random keys $\Delta, K[x_1], \ldots, K[x_n] \in_R \mathbb{Z}_p$ are chosen by $\mathcal{V}$, and then $\mathcal{P}$ obtains the MACs $M[x_i] \leftarrow \Delta \cdot x_i + K[x_i] \in \mathbb{Z}_p$. We use the notation $[x_i]_p$ for this. To open $[x]_p$, $\mathcal{P}$ sends x and $M[x]$ to $\mathcal{V}$, who checks that $M[x] = \Delta \cdot x + K[x]$ holds. These authentications are linearly homomorphic: Given authenticated values $[x]_p$ and $[y]_p$ and public values a, b, $\mathcal{P}$ and $\mathcal{V}$ can locally compute $[z]_p$ for $z := a \cdot x + y + b$ by setting $M[z] := a \cdot M[x] + M[y]$ and $K[z] := a \cdot K[x] + K[y] - \Delta \cdot b$. For large enough p, this is secure since forgery would imply correctly guessing a random element of $\mathbb{Z}_p$. For smaller p, the keys Δ and $K[x_i]$ are instead chosen from an extension field $\mathbb{Z}_{p^r}$ such that p^r is large enough. The MACs can be efficiently computed with (subfield) VOLE [4, 31].

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 [31] and Mac'n'Cheese [4]. We refer to their works for details. For rings $\mathbb{Z}_{2^k}$, we present an instantiation in Section 5.2.

We formally define the homomorphic commitments using the ideal functionality $\mathcal{F}^R_{\mathsf{ComZK}}$ given in Appendix A, Figure 15. The parameter R denotes the message space, which is in our case either a ring $\mathbb{Z}_{2^k}$ or a field $\mathbb{Z}_p$. In addition to the common Input and Open operations, which enables $\mathcal{P}$ to commit to a value and reveal it to $\mathcal{V}$ at a later point, we also model Random and CheckZero, for generating commitments of random values and verifying that a committed value equals zero, respectively, which enables more efficient implementations. Moreover, $\mathcal{F}^R_{\mathsf{ComZK}}$ allows via Affine to compute affine combinations of committed values with public coefficients yielding again a commitment of the result. Finally, CheckMult allows to verify that a set of committed triples satisfy a multiplicative relation, i.e. for each triple, the third commitment contains the product of the first two committed values.

Since the commitment functionality is based on information-theoretic MACs, we use the same notation $[x]$ to denote a committed value $x \in R$. We use this shorthand to to simplify the presentation of higher-level protocols without explicitly mentioning the commitment identifiers. We use also shorthands for the different methods of $\mathcal{F}^R_{\mathsf{ComZK}}$, e.g. we write something like $[z] \leftarrow a \cdot [x] + [y] + b$ when invoking the Affine method. 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.

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 [26, 28].

Their generalization, called `edaBits` (due to Escudero *et al.* [16]), is defined as m bits $b_0, \ldots, b_{m-1}$ which are each authenticated in the binary domain while their sum is authenticated in the arithmetic one, i.e. $([b_0]_2, \ldots, [b_{m-1}]_2, [b]_M)$, for some $m \leq \lceil \log M \rceil$. These `edaBits` allow for optimized 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 [16] consists of two different phases: in the first phase, each party locally samples `edaBits` and proves to all other parties that they were computed correctly. Then, in a second phase, these local contributions are combined to global, secret `edaBits`. In our setting however, only the prover will use `edaBits`, thus it is clear that the second phase can be omitted. Our sampling protocol will only have to ensure that each `edaBit` $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ is indeed consistent, i.e. that $x = \sum_{i=0}^{m-1} x_i 2^i \bmod M$.

The first phase of the `edaBit` sampling routine of [16] then works as follows (when adapted to the zero-knowledge setting):

(1) The prover locally samples $(NB+C)m$ bits $r_{i,j}$ for $j \in [NB+C]$ and $i \in \{0, \ldots, m-1\}$. It then combines these into the $NB+C$ values $r_j \leftarrow \sum_{i=0}^{m-1} 2^i r_{i,j}$ yielding $\texttt{edaBits}\,\{(r_{i,j})_{i=0}^{m-1}, r_j\}_{j \in [NB+C]}$.
(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; [16] show that certain consistency checks can be performed in an unreliable manner, while still being hard to cheat overall, which leads to a complicated analysis.

3 CONVERSIONS BETWEEN $\mathbb{Z}_2$ AND $\mathbb{Z}_M$

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

3.1 Conversions and edaBits in ZK

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

Definition 3.1 (Conversion Tuple). Let $M \in \mathbb{N}^+$, $m \leq \lceil \log_2(M) \rceil$, $x \in \mathbb{Z}_M$ and $x_i \in \mathbb{Z}_2$. Then the tuple $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ is called a *conversion tuple*. We call $([x_0]_2, \ldots, [x_{m-1}]_2, [x]_M)$ consistent iff $x = \sum_{i=0}^{m-1} 2^i x_i \bmod M$.

Our conversion protocol in this section provides an efficient way to verify that a large batch of conversion tuples are consistent, i.e. that the committed values are indeed valid edaBits. We note that an alternative approach would be to directly apply the method of [16] — here, first a set of *random*, verified conversion tuples is created, and then one of these is used to check the actual conversion tuple in an online phase. Unfortunately, this online phase check itself involves verifying a binary circuit for addition mod M, which introduces additional expense. We therefore design a new protocol to avoid this, with further optimizations.

Our protocols perform conversions on committed values in $\mathbb{Z}_2$ and $\mathbb{Z}_M$, where we recall that M is either a large prime or 2^k. We model these commitments using the functionality $\mathcal{F}_{\text{ComZK}}^{2,M}$ in Figure 16in Appendix A, which extends two instances of $\mathcal{F}_{\text{ComZK}}^{R}$ for $R = \mathbb{Z}_2$ and $R = \mathbb{Z}_M$ and simply parses all method calls to the respective instance.

Finally, we define the ideal functionality for verifying conversions $\mathcal{F}_{\text{Conv}}$ in Figure 1. This functionality extends $\mathcal{F}_{\text{ComZK}}^{2,M}$ with a single method VerifyConv. It essentially checks whether or not the two representations of some hidden value are consistent.

3.2 The Conversion Verification Protocol Π_{Conv}

The following protocol Π_{Conv} verifies the correctness of a batch of N conversion tuples. Π_{Conv} uses $\mathcal{F}_{\text{Dabit}}$ (Figure 2) to verify correctness of daBits (recall, a daBit is an edaBit of length 1), which is needed in one stage of the protocol. Later, we show how to remove most of the daBit check to improve efficiency.

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

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

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

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

Figure 1: Functionality $\mathcal{F}_{\text{Conv}}$ checking edaBits

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

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

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

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

Figure 2: Functionality $\mathcal{F}_{\text{Dabit}}$ checking daBits.

On a high level, Π_{Conv}, in Figure 3, consists of three phases:

(1) Initially, $\mathcal{P}$ commits to auxiliary random edaBits, daBits and multiplication triples necessary for the check. The daBits are verified separately, and then $\mathcal{V}$ chooses a random permutation.
(2) After permuting the edaBits and multiplication triples, both parties run an implicit cut-and-choose phase. Here, $\mathcal{P}$ opens C of the edaBits and triples, which are checked by $\mathcal{V}$.
(3) We place each conversion tuple into one of N buckets, each of which contains a conversion tuple $([c_0]_2, \ldots, [c_{m-1}]_2, [c]_M)$, and a set of B edaBits $\{([r_0]_2, \ldots, [r_{m-1}]_2, [r]_M)_i\}_{i=0}^{B-1}$. None of these have been proven consistent, but C edaBits coming from the same pool have been opened in the previous step. Now, over B iterations the prover and verifier for each $j \in [B]$ compute $[c + r_j]_M = [c]_M + [r_j]_M$ and use an addition circuit to check that $([e_0]_2, \ldots, [e_m]_2) = ([c_0]_2, \ldots, [c_{m-1}]_2) + ([r_0]_2, \ldots, [r_{m-1}]_2)$. The addition circuit is evaluated using the multiplication triples (which also may be inconsistent).

For the checks within each bucket, we use the two sub-protocols convertBit2A (Figure 4) and bitADDcarry (Figure 5). 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 [16].

Protocol Π_{Conv}

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

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

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

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

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

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

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

Figure 3: Protocol Π_{Conv} to verify Conversion Tuples

Definition 3.2. A binary circuit $C : \mathbb{Z}_2^{2m} \to \mathbb{Z}_2^{m+1}$ is weakly additively tamper resilient, if given any additively tampered circuit C^*, obtained by flipping the output of any fixed number of AND gates in C, one of the following two properties hold:

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

Procedure convertBit2A

Input A daBit $([r]_2, [r]_M)$ and a commitment $[x]_2$.
Protocol
(1) $c \leftarrow$ Open($[r]_2 \oplus [x]_2$).
(2) Output $[x]_M \leftarrow c + [r]_M - 2 \cdot c \cdot [r]_M$.

Figure 4: Procedure to convert bit from $\mathbb{Z}_2$ to $\mathbb{Z}_M$.

Procedure bitADDcarry

Input Commitments $[x_0]_2, \ldots, [x_{m-1}]_2, [y_0]_2, \ldots, [y_{m-1}]_2$.
Protocol Let $c_0 = 0$.
(1) Compute $[c_{i+1}]_2 = [c_i]_2 \oplus (([x_i \oplus c_i]_2) \wedge ([y_i \oplus c_i]_2)), \forall i \in \{0, \ldots, m-1\}$
(2) Output $[z_i]_2 = [x_i \oplus y_i \oplus c_i]_2, \forall i \in \{0, \ldots, m-1\}$ and $[c_m]_2$.

Figure 5: A ripple-carry adder

Note that the type of additive tampering in Definition 3.2 models the errors induced by faulty multiplication triples, when used to evaluated a circuit in ZK or MPC. Intuitively, the definition says that the output of the tampered circuit is either incorrect on every possible input or equivalent to the original un-tampered circuit. This gives us the property that an adversary cannot pass the verification protocol using a tampered circuit with both a good conversion tuple and a bad one. Thus, if any provided multiplication triples are incorrect, then the check at those positions would only pass with either a good or a bad conversion tuple (or edaBit), but not both.

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

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

Our implementation shows that our approach outperforms or is competitive with all prior work. We discuss the implementation and the concrete performance in Section 6.

3.3 Proof of security

Due to space constraints, the full proof of security can be found in the full version [3]. We summarize the proof below.

In order to prove the security of Π_{Conv}, we first observe that instead of letting $\mathcal{P}$ choose multiplication triples, we might equivalently model this by letting $\mathcal{P}$ specify circuits instead (that will be evaluated instead of the Ripple Carry Adder). Then, we define an

abstraction of the protocol as a balls-and-bins type game, similar to [16], and analyze the success probability of an adversary in this game.$\mathcal{A}$ winning in this abstraction rather than in the protocol Π_{Conv}. We make this abstraction, as a straightforward analysis of the conversion protocol is rather complex. This is due to there being multiple ways for $\mathcal{A}$ to pass the check with a bad conversion tuple. The first is by corrupting K conversion tuples, then corrupting $K \cdot B$ edaBits and hoping that these end up in the right buckets, canceling out the errors in the conversion tuples. The second approach is to corrupt a set of edaBits and then guess the arrangement of these, thus yielding how many circuits $\mathcal{A}$ would have to corrupt in order to cancel out the errors of the conversion tuples. Furthermore, conversion tuples (and edaBits) may be corrupted in several ways. To avoid these issues, we describe an abstract security game which only provides a better chance for the adversary to win than the original protocol. In summary, we show the following:

Theorem 3.3. *The probability of Π_{Conv} not detecting at least one incorrect conversion tuple is upper bounded by 2^{-s} whenever $N \geq 2^{s/(B-1)}$ and $C = C' = B$ for bucket size $B \in \{3, 4, 5\}$.*

The proof can be found in the full version [3]. The approach is similar to that of [16], 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 the full version [3] we then prove security of Π_{Conv}:

Theorem 3.4. *Let $N \geq 2^{s/(B-1)}$, $C = C' = C'' = B$ and $B \in \{3, 4, 5\}$ such that $\frac{s}{B-1} > B$, then protocol Π_{Conv} (Figure 3) UC-realises $\mathcal{F}_{\text{Conv}}$ (Figure 1) in the $\mathcal{F}_{\text{Dabit}}$-hybrid model. Specifically, no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most 2^{-s}.*

3.4 Faulty daBits

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

Definition 3.5. A *faulty* daBit is a pair $([b]_2, [b']_M)$ such that $b \equiv b' \bmod 2$, but not necessarily $b' \in \{0, 1\}$.

In Step 6 of Π_{Conv} (Figure 3), daBits are used in convertBit2A (Figure 4) to transform the final carry bit $[e_m]_2$ from bitADDcarry (Figure 5) into $[e'_m]_M$ such that $e_m = e'_m$. We show that using a faulty daBit cannot help the adversary in passing the check with incorrect conversions. First, we observe that with a faulty daBit, the output becomes $e'_m = e_m + (-1)^{e_m} \cdot \delta$ where $\delta = (-1)^b \cdot (b' - b)$ where $(b' - b) > 1$ represents the difference (or error) between b' and b for the used daBit $([b]_2, [b']_M)$. As a result, $|e'_m| > 1$ where $|\cdot|$ denotes absolute value. This carry bit $[e'_m]_M$ is then used to remove the potential carry from $[c + r]_M$ by computing

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

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

A full security analysis, showing that faulty daBits do not impact the security of Π_{Conv}, can be found in the full version [3]. There, we define a functionality $\mathcal{F}_{\text{FDabit}}$ that encompasses this idea of two bits (b, b') where $b' \equiv b \bmod 2$.

Creating Faulty daBits. We now describe a revised daBit verification protocol Π_{FDabit} realizing $\mathcal{F}_{\text{FDabit}}$, that, for a given daBit $([b]_2, [b']_M)$, only verifies that $b \equiv b' \bmod 2$. For this check, it suffices to compute random linear combinations in the two domains and then open the values, giving consistency modulo 2. We define the protocol Π_{FDabit} in Figure 6, and show the following statement in the full version [3]:

Lemma 3.6. *Protocol Π_{FDabit} (Figure 6) UC-realizes $\mathcal{F}_{\text{FDabit}}$ except with probability 2^{-s+1}.*

Protocol Π_{FDabit}

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

Protocol Perform the following check s times:

(1) $\mathcal{P}$ samples γ random bits $\{[c_i]_p\}_{i \in [\gamma]}$. It additionally creates $[c_1]_2$. It does not prove consistency among $[c_1]_p$ and $[c_1]_2$.

(2) $\mathcal{P}$ shows that $\{[c_i]_p\}_{i \in [\gamma]}$ are bits by showing that 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 $[r]_2 \leftarrow [c_1]_2 \oplus \bigoplus_{i=1}^{N} e_i \cdot [b_i]_2$

(5) $r \leftarrow$ Open$([r]_2)$

(6) Let $[r']_p \leftarrow [\sum_{i=1}^{N} e_i \cdot b_i]_p$

(7) Let $\tau =$ Open$([r']_p + \sum_{j=1}^{\gamma} [c_i]_p \cdot 2^{i-1})$.

(8) Check if $0 \leq \tau < 2^{\gamma}$ and $r = \tau \bmod 2$. If not, abort. Output (success).

Figure 6: Our optimized consistency check for daBits, that no longer checks that $[b]_p$ is a bit

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 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.

4 TRUNCATION AND INTEGER COMPARISON

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

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

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

4.1 Truncation

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

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

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

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

Figure 7: Functionality $\mathcal{F}_{\text{VerifyTrunc}}$ that verifies a truncation

In Figure 7 we present a functionality $\mathcal{F}_{\text{VerifyTrunc}}$ that takes a batch of commitments $[a_j]_M$ and their supposed truncations (by m^j bits) $[a'_j]_M$. The functionality ensures that the truncations are correct, namely, $a'_j = \lfloor \frac{a_j}{2^{m^j}} \rfloor$. Note that this functionality we realise is flexible, in that it can support a large batch of truncations, each of which may be of a different length.

We now construct a protocol for verifying truncations, which can securely realise $\mathcal{F}_{\text{VerifyTrunc}}$ using just a *single* call to our batch conversion functionality, $\mathcal{F}_{\text{Conv}}$, 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 $([a_{tr}^0]_2, \ldots, [a_{tr}^{l-m-1}]_2)$
- the initial m bits of $[a]_M$; $[a']_M = [a \mod 2^m]_M$ as well as its bit decomposition $([a'_0]_2, \ldots, [a'_{m-1}]_2)$

Having access to $[a_{tr}]_M$ and $[a']_M$ allows the verifier then to check that $a = 2^m \cdot a_{tr} + a'$, which is sufficient to prove the claim.Observe that running Π_{Conv} on $[a_{tr}]_M$ and $([a_{tr}^0]_2, \ldots, [a_{tr}^{l-m-1}]_2)$ not only shows consistency between the binary and arithmetic representations, but also that $[a_{tr}]_M$ can be represented by $l-m$ or less bits (same goes for $[a']_M$ and its bit decomposition).

We first define an ideal functionality $\mathcal{F}_{\text{CheckLength}}$ (Figure 8) that encapsulates this concept of using $\mathcal{F}_{\text{Conv}}$ 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 $\mathcal{F}_{\text{Conv}}$. The cost of the protocol also directly follows from the consistency check described in Figure 3.

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

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

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

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

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

Figure 8: Functionality to verify length of commitments

4.2 Integer Comparison

We now discuss how to compare two signed, l-bit integers α and β. The way the protocol works is by having the prover (and verifier) compute $[\alpha]_M - [\beta]_M$ and have the prover compute the truncation of this which is only the most significant bit. Now we may run $\Pi_{\text{VerifyTrunc}}$ on the truncation and use the truncation as the output of the comparison. We remark that, similarly to previous works in the MPC setting [11, 16], this gives the correct result as long as

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

Input A set of tuples $\{[x^j]_M, m^j, \}_{j\in[N]}$ where $x^j \in [0, 2^l)$ and m^j defines the claimed bitlength of x^j.
Protocol
(1) For each $j \in [N]$, $\mathcal{P}$ commits to $[x_0^j]_2, \ldots, [x_{m^j-1}^j]_2$.
(2) Let $m = \max_j\{m^j\}$, and for $i = m^j, \ldots, m-1$, let $[x_i^j]_2$ denote a dummy commitment to zero (which can be easily obtained with CheckZero).
(3) Run $\mathcal{F}_{\mathsf{Conv}}$ on $\{([x_0^j]_2, \ldots, [x_{m-1}^j]_2, [x^j]_M)\}_{j\in[N]}$ and output what $\mathcal{F}_{\mathsf{Conv}}$ outputs.

Figure 9: Protocol $\Pi_{\mathsf{CheckLength}}$ that verifies that committed elements are bounded.

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

Input A set of tuples $\{[a^j]_M, m^j, [a_{tr}^j]_M\}_{j\in[N]}$ where $a^j \in [0, 2^l)$, m^j defines the number of bits that has been truncated and $[a_{tr}]_M$ represents the supposed truncation.
Protocol
(1) For each $j \in [N]$, $\mathcal{P}$ commits to the least-significant m bits of $[a^j]_M$, denoted as $[a']_M = [a^j \bmod 2^m]_M$.
(2) The parties call $\mathcal{F}_{\mathsf{CheckLength}}$ with input $\{[a']_M, m^j\}_{j\in[N]} \cup \{[a_{tr}^j]_M, l - m^j\}_{j\in[N]}$.
(3) For each j, let $[y]_M = [a^j]_M - (2^m \cdot [a_{tr}^j]_M + [a']_M)$ and run $\mathsf{CheckZero}([y]_M)$.
Abort if any of the checks fail. Otherwise output (success).

Figure 10: Protocol to verify the truncation of an element from $\mathbb{Z}_M$

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

5 INTERACTIVE PROOFS OVER $\mathbb{Z}_{2^k}$

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

5.1 Linearly Homomorphic Commitments from Vector-OLE

To construct linearly homomorphic commitments over the ring $\mathbb{Z}_{2^k}$, we use a variant of the information-theoretic MAC scheme from SPD$\mathbb{Z}_{2^k}$ [12]: Let s be a statistical security parameter. To authenticate a value $x \in \mathbb{Z}_{2^k}$ known to $\mathcal{P}$ towards $\mathcal{V}$ (denoted as $[x]$), we choose the MAC keys $\Delta \in_R \mathbb{Z}_{2^s}$ and $K[x] \in_R \mathbb{Z}_{2^{k+s}}$, and compute the MAC tag as

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

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

Initially $\tilde{x}$ may be chosen as $\tilde{x} = x \in \{0, \ldots, 2^k - 1\}$. 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 held by $\mathcal{P}$.

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

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

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

As in SPD$\mathbb{Z}_{2^k}$, the MACs are obtained using vector OLE over rings. We describe the protocols in the $\mathcal{F}_{\mathsf{vole2k}}^{s,r}$-hybrid model (cf. Figure 11); in Section 5.4, we discuss how to instantiate this VOLE functionality. To open a commitment $[x]$, first the upper s bits of $\tilde{x}$ need to be randomized, by computing $[z] \leftarrow [x] + 2^k \cdot [r]$ with random $\tilde{r} \in_R \mathbb{Z}_{2^{k+s}}$. Then, $\tilde{z}$ and $M[z]$ are published and the MAC equation (Equation (1)) is verified. Following [13, 31], we implement more efficient batched checks based on a random oracle in protocol $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ (Figures 12 & 13) and protocol $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ (Figure 14).

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

Init This method needs to be the first one called by the parties. On input (Init) from both parties the functionality
(1) If $\mathcal{V}$ is honest, it samples $\Delta \in_R \mathbb{Z}_{2^s}$ and sends Δ to $\mathcal{V}$.
(2) If $\mathcal{V}$ is corrupt, it receives $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{S}$.
(3) Δ is then stored by the functionality.
All further Input queries are ignored.

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

Figure 11: Functionality for vOLE with key size s and message size r. Based on $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ from [31, Fig. 2].

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

In this section, we present two protocols $\Pi_{\text{ComZK-a}}^{\mathbb{Z}_{2^k}}$ and $\Pi_{\text{ComZK-b}}^{\mathbb{Z}_{2^k}}$ which instantiate the $\mathcal{F}_{\text{ComZK}}^{\mathbb{Z}_{2^k}}$ functionality (Figure 15). These are adaptions of the Wolverine [31] and Mac'n'Cheese [4] protocols to the $\mathbb{Z}_{2^k}$ setting and differ mainly in the implementation of the CheckMult method.

For CheckZero, we use in both variants the batched check from [13, 31] based on a random oracle $H\colon \{0,1\}^* \to \{0,1\}^s$: First, the upper s bits (resp. the upper $2s$ bits in $\Pi_{\text{ComZK-b}}^{\mathbb{Z}_{2^k}}$) of each value $[x_i]$ are randomized by computing $[y_i] \leftarrow [x_i] + 2^k \cdot [r_i]$ with random r_i. Then $\mathcal{P}$ sends the upper bits p_i of all the $\tilde{y}_i$ and a hash $h := H(M[y_1], \ldots, M[y_n])$ to $\mathcal{V}$. Finally, $\mathcal{V}$ uses the p_i to recompute the MAC tags $M[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + K[y_i]$ and verifies that $h \stackrel{?}{=} H(M[y_1]', \ldots, M[y_n]')$ holds.

A previous version of this paper used a version of the batched check described in SPD$\mathbb{Z}_{2^k}$ [12] based on a random linear combination. This would have been more efficient since it does not require sending the randomized upper bits of each value separately. Unfortunately, due to a bug in their proof this check is not sound, so we cannot use it here. A less efficient adaption of the check could be used if one wants to avoid using a random oracle, though.

$\Pi_{\text{ComZK-a}}^{\mathbb{Z}_{2^k}}$ (Figures 12 & 13) adapts the bucketing approach from Wolverine [31]: Let $C, B \in \mathbb{N}$ be the parameters of the bucketing scheme. To check that a collection of triples $([a_i], [b_i], [c_i])_{i=1}^n$ satisfy a multiplicative relation, i.e. $a_i \cdot b_i = c_i$ for $i = 1, \ldots, n$, the prover creates a set of $\ell := n \cdot B + C$ unchecked multiplication triples of commitments. After randomly permuting the ℓ triples according to the choice of the verifier, C triples are opened and checked by the verifier. The remaining nB triples are evenly distributed into n buckets. Then, each multiplication $(a_i \cdot b_i \stackrel{?}{=} c_i)$ is verified with the B triples of the corresponding bucket with a variant of Beaver's multiplication trick [5]. 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 $\Pi_{\text{ComZK-b}}^{\mathbb{Z}_{2^k}}$ (Figure 14), we have adapted the multiplication check of Mac'n'Cheese [4], which is similar to the Wolverine [31] optimization for large fields and SPDZ-style [14] sacrificing of multiplication triples. The soundness of this type of check is based on the difficulty of finding a solution to a randomized equation. If a multiplicative relation does not hold, the adversary needs to guess a random field element in order to pass. Thus the original scheme needs a large field to be sound. In the $\mathbb{Z}_{2^k}$-setting, there are multiple obstacles that we have to overcome. First, we would like to also support small values of k (e.g. $k = 8$ or 16). Simultaneously, we also have to deal with zero divisors (which complicate the check) which were no issue in the field setting. Moreover, even though the commitment scheme (see Section 5.1) uses the larger ring $\mathbb{Z}_{2^{k+s}}$ it only authenticates the lower k bits of $\tilde{x}$ and cannot prevent modifications of the upper bits, which might lead to additional problems. We overcome these challenges by further increasing the ring size from $\mathbb{Z}_{2^{k+s}}$ to $\mathbb{Z}_{2^{k+2s}}$, so that the commitment scheme provides authenticity of values modulo 2^{k+s}. We use the additional s bits to avoid overflows when checking correctness of the multiplicative relations modulo 2^k with an s bit random challenge. Increasing the ring leads to larger storage and communication requirements – the values $\tilde{x}, M[x], K[x]$ now require $k + 2s$ bits. We discuss the communication complexity of both variants in Section 6.1.1.

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, the complete proofs are given in Appendix E.

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

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

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

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

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

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

CheckZero Let $H\colon \{0,1\}^* \to \{0,1\}^s$ denote a random oracle. For (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 $[r_1], \ldots, [r_n] \leftarrow$ Random() and compute $[y_i] \leftarrow [x_i] + 2^k \cdot [r_i]$ for $i = 1, \ldots, n$.
(3) $\mathcal{P}$ sends $p_1, \ldots, p_n$ to $\mathcal{V}$ where $p_i := (\tilde{y}_i - y_i)/2^k$ denotes the upper s bits of $\tilde{y}_i$.
(4) $\mathcal{P}$ computes $h \leftarrow H(M[y_1], \ldots, M[y_n])$ and sends $h \in \{0,1\}^{2\lambda}$ to the verifier.
(5) Finally, $\mathcal{V}$ computes $M[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + K[y_i] \in \mathbb{Z}_{2^{k+s}}$ for $i = 1, \ldots, n$, checks $h \stackrel{?}{=} H(M[y_1]', \ldots, M[y_n]')$ and outputs (success) if the equality holds and aborts otherwise

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

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

Figure 12: Protocol $\Pi_{\text{ComZK-a}}^{\mathbb{Z}_{2^k}}$ instantiating $\mathcal{F}_{\text{ComZK}}^{\mathbb{Z}_{2^k}}$ using a Wolverine-like [31] multiplication check.

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

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) $\mathcal{P}$ aborts if $a_i \cdot b_i \neq c_i \pmod{2^k}$ 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 $\mathcal{P}$ receives $(x_i, y_i)_{i=1}^{\ell}$.

(4) $\mathcal{P}$ computes $z_i \leftarrow x_i \cdot y_i$ for $i = 1, \ldots, \ell$, and they run $([z_i])_{i=1}^{\ell} \leftarrow \text{Input}((z_i)_{i=1}^{\ell})$.

(5) $\mathcal{V}$ samples a permutation $\pi \in_R S_\ell$ and sends it to $\mathcal{P}$.

(6) They run $(x_{\pi(i)}, y_{\pi(i)}, z_{\pi(i)})_{i=1}^{C} \leftarrow \text{Open}(([x_{\pi(i)}], [y_{\pi(i)}], [z_{\pi(i)}])_{i=1}^{C}, \text{lst})$.

(7) $\mathcal{V}$ 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) $d \leftarrow \text{Open}([a_j] - [x_{\pi(k)}], \text{lst})$ and $e \leftarrow \text{Open}([b_j] - [y_{\pi(k)}], \text{lst})$ and
   (b) $[w_k] \leftarrow [z_{\pi(k)}] - [c_j] + e \cdot [x_{\pi(k)}] + d \cdot [y_{\pi(k)}] + d \cdot e$

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

Figure 13: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\text{ComZK}}$ using a Wolverine-like [31] multiplication check.

5.3.1 Proof of $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$. We state the security of our protocol as follows:

Theorem 5.1. *The protocol $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$ (Figures 12 & 13) securely realizes the functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\text{ComZK}}$: No environment can distinguish the real execution from a simulated one except with probability $(q_{\text{cz}} + q_{\text{cm}}) \cdot 2^{-s+1} + q_{\text{cm}} \cdot \binom{nB+C}{B}^{-1}$, 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 $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$. These are stated in the following two lemmata. The full proof of Theorem 5.1 is given in Appendix E.1.

Lemma 5.2. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckZero *protocol of $\Pi^{\mathbb{Z}_{2^k}}_{\text{com-a}}$ with commitments $[x_1], \ldots, [x_n]$ and $x_i \not\equiv_k 0$ for some $i \in \{1, \ldots, n\}$ then $\mathcal{V}$ outputs* (success) *with probability at most $\varepsilon_{\text{cz}} := 2^{-s+1}$.*

The CheckZero protocol is based on the batch check from [13, 31], and the proof of Lemma 5.2 is given in Appendix E.2.

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

Much of the protocol is identical to $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$ (Figures 12 and 13) with the exception that the MACs are now computed in the larger ring $\mathbb{Z}_{2^{k+2s}}$. **Init**, **Random**, **Affine Combination**, **Input** and **Open** work exactly as in $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$, although using $\mathcal{F}^{s,k+2s}_{\text{vole2k}}$.

CheckZero Let $H\colon \{0,1\}^* \to \{0,1\}^s$ denote a random oracle. 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 $[r_1], \ldots, [r_n] \leftarrow \text{Random}()$ and compute $[y_i] \leftarrow [x_i] + 2^k \cdot [r_i]$ for $i = 1, \ldots, n$.

(3) $\mathcal{P}$ sends $p_1, \ldots, p_n$ to $\mathcal{V}$ where $p_i := (\tilde{y}_i - y_i)/2^k$ denotes the upper $2s$ bits of $\tilde{y}_i$.

(4) $\mathcal{P}$ computes $h \leftarrow H(M[y_1], \ldots, M[y_n])$ and sends $h \in \{0,1\}^{2\lambda}$ to the verifier.

(5) Finally, $\mathcal{V}$ computes $M[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + K[y_i] \in \mathbb{Z}_{2^{k+2s}}$ for $i = 1, \ldots, n$, checks $h \stackrel{?}{=} H(M[y_1]', \ldots, M[y_n]')$ and outputs (success) if the equality holds and aborts otherwise

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

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

(1) $\mathcal{P}$ aborts if $a_i \cdot b_i \neq c_i \pmod{2^k}$ for some $i = 1, \ldots, n$.

(2) Let $\text{lst} := \emptyset$.

(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 $\varepsilon_i \leftarrow \text{Open}'(\eta \cdot [a_i] - [x_i], \text{lst})$ for $i = 1, \ldots, n$.

(6) Run $\text{CheckZero}'((\eta \cdot [c_i] - [z_i] - \varepsilon_i \cdot [b_i])_{i=1}^n, \text{lst})$. If successful, $\mathcal{V}$ returns (success), otherwise abort.

Figure 14: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\text{ComZK}}$ using a Mac'n'Cheese-style [4] multiplication check.

Lemma 5.3. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckMult *protocol of $\Pi^{\mathbb{Z}_{2^k}}_{\text{com-a}}$ with parameters $B, C \in \mathbb{N}$ such that $C \geq B$ and inputs $([a_i], [b_i], [c_i])_{i=1}^n$ and there exists an index $1 \leq i \leq n$ such that $a_i \cdot b_i \not\equiv_k c_i$ then $\mathcal{V}$ outputs* (success) *with probability at most $\varepsilon_{\text{cm}} + \varepsilon_{\text{cz}}$ with $\varepsilon_{\text{cm}} := \binom{nB+C}{B}^{-1}$, and ε_{cz} the soundness error of* CheckZero *given in Lemma 5.2.*

The CheckMult protocol is based on the corresponding check from Wolverine [31], and the same analysis also applies to the $\mathbb{Z}_{2^k}$ case. The proof of Lemma 5.3 can be found in Appendix E.3.

5.3.2 Proof of $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$. The formal statement of security is given in the following theorem:

Theorem 5.4. *The protocol* $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$ *(Figure 14) securely realizes the functionality* $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\text{ComZK}}$*: No environment can distinguish the real execution from a simulated one except with probability* $(q_{\text{cz}} + q_{\text{cm}}) \cdot 2^{-s+1} + q_{\text{cm}} \cdot 2^{-s}$*, where* q_{cz} *is the sum of calls to* CheckZero *and* Open*, and* q_{cm} *the number of calls to* CheckMult.

The proof of Theorem 5.4 is given in Appendix E.4. Except for the simulation of CheckMult, it is largely similar to the proof of Theorem 5.1. Again, we first prove a lemma about the soundness error of the CheckMult operation, that we use to show indistinguishability of our simulation.

Lemma 5.5. *If* $\mathcal{P}^*$ *and* $\mathcal{V}$ *run the* CheckMult *protocol of* $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$ *with inputs* $([a_i], [b_i], [c_i])_{i=1}^n$ *such that there exists an index* $1 \le i \le n$ *such that* $a_i \cdot b_i \not\equiv_k c_i$*, then* $\mathcal{V}$ *outputs* (success) *with probability at most* $\varepsilon'_{\text{cm}} + \varepsilon_{\text{cz}}$ *with* $\varepsilon'_{\text{cm}} := 2^{-s}$*, and* ε_{cz} *the soundness error of* CheckZero *given in Lemma 5.2.*

The proof of Lemma 5.5 is given in Appendix E.5.

5.4 Instantiating VOLE mod 2^k

Our ZK protocol over $\mathbb{Z}_{2^k}$ requires an actively secure protocol for VOLE in $\mathbb{Z}_{2^{k+s}}$. Unfortunately, this means we cannot take advantage of the most efficient LPN-based protocols [8, 31], which currently only have an actively secure setup protocol over fields. We consider two possible alternatives. First, as done in [12], we can use the protocol for correlated oblivious transfer over general rings from [29], which gives an amortized communication cost of $s(k+s)$ bits per VOLE. This is quadratic in the bit length, which will be a bottleneck for our ZK protocols in terms of communication.

Alternatively, we can obtain sublinear communication using LPN-based VOLE, but using generic actively secure 2-PC for the setup. Here, we can use either the primal variant of LPN over rings, as done in [30], or dual-LPN based on quasi-cyclic codes, as used over $\mathbb{Z}_2$ in [8] (these can also be defined over $\mathbb{Z}_{2^k}$ under an analogous hardness assumption). Since dual-LPN has lower communication, in the following we assume this variant. Now, for the setup procedure, if we produce a VOLE of length $N = 10^7$ with parameters $(c, t) = (4, 54)$ from [8], the bottleneck is around $2t \log(cN/t)$ AES evaluations in 2-PC, which gives a total of ≈ 1 AND gate per VOLE output. Using a TinyOT-like protocol [20] combined with LPN-based OT [8, 34], each AND gate needs around 32 bits of communication, more than an order of magnitude less than the first approach (note that TinyOT incurs a much larger round complexity).

For future work, an important problem is to adapt the current techniques for actively secure VOLE over fields to the ring setting, which would greatly reduce the preprocessing cost.

Table 1: Amortized communication cost in bits per instruction. k **is the size of the modulus,** s **depends on the statistical security parameter,** B **is the bucket size of** $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$**.**

Protocol	CheckZero	Open	CheckMult
$\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$	s & 1 VOLE	$k+s$ & 1 VOLE	$3B(k+s)$ & $4B$ VOLE
$\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$	$2s$ & 1 VOLE	$k+2s$ & 1 VOLE	$2k+4s$ & 3 VOLE

6 EVALUATION

6.1 Communication Complexity

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

As described in Section 5.1 and Section 5.2, we need to randomize the upper s or $2s$ bits when doing a CheckZero or Open operation. We note that in $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$ Step 8a and $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$ Step 5 the values get already masked with uniformly random values directly before the Open operation. Hence, the extra masking step during Open can be omitted. A similar optimization can be applied in Π_{Conv}.

The amortized communication costs per checked commitment and multiplication triple of both protocols are given in Table 1.

6.1.2 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 2, in terms of the amount of communication required, and preprocessed correlated OTs or VOLEs. Note that to simplify the table, we assume that $m \approx \log M$, and so count the cost of sending one $\mathbb{Z}_M$ element in the protocol as m bits. Also, in this analysis we ignore costs that are independent of the number of conversions being checked, such as the small number of checks in the faulty `daBit` protocol. In Appendix D, we give a more detailed breakdown of these costs, including complexities of the sub-protocols bitADDcarry and convertBit2A.

The "naïve" way of verifying the a conversion would be to have the prover provide both a set of bits $\chi = \{[x_0]_p, \ldots, [x_{m-1}]_p\}$ as well as the value $[x]_p$ and then verify that each element in χ is in fact a bit, as well as that they sum to the value $[x]_p$. This requires sampling m random VOLEs as well as fixing each of these to a value chosen by the prover. Afterwards the prover proves that each is a bit by computing $\text{CheckZero}([x_i]_p \cdot ([x_i]_p - 1))$, $[x_i]_p \in \chi$ which requires multiplication triples over $\mathbb{F}_p$ as well as communication. We list the cost of this "naïve" way of verifying the conversion Table 2. To verify the multiplications we use the basic version of Mac'n'Cheese [4]. The "basic" baseline comparison in Table 2 comes from a straightforward application of using `edaBits` for ZK, similarly to [16]. Namely, this protocol would first generate consistent `edaBits` using [16], and then verify the conversion using a single binary addition circuit (similar to the bucket-check in Figure 3, step 6). However, this requires doing the check with m *verified* multiplication triples (over $\mathbb{Z}_2$) and a single `daBit`, which in turn requires an additional verified multiplication (over $\mathbb{Z}_M$). To

Table 2: Costs of verifying conversions between $\mathbb{Z}_2$ and $\mathbb{Z}_M$ in terms of COTs, VOLEs, and additional communication. The "basic" protocol uses `edaBits` directly, while "Section 3" uses our optimizations. "QS-Circuit" and "QS-Poly" refer to variants that use [33] for circuits and sets of polynomials respectively. $m \approx \log(M), k$ denotes the bitsize of the converted value, and B is the bucket size. For $\mathbb{Z}_{2^k}$, the costs for $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-a}}$ are given. The $\Pi^{\mathbb{Z}_{2^k}}_{\text{ComZK-b}}$ variant requires Bs additional bits of communication (see Section 5.2).

Protocol	Comm. in bits	#COTs	#VOLEs
naive, $\mathbb{Z}_p$	$2m^2$	0	$2m$
basic, $\mathbb{Z}_p$, $\log(p) \leq s$	$13Bm + 6m + B - 1$	$4Bm + 3m + B - 1$	$11B - 4$
basic, $\mathbb{Z}_p$, $\log(p) > s$	$10Bm + 6m + B - 1$	$4Bm + 3m + B - 1$	$8B - 4$
Section 3, $\mathbb{Z}_p$	$6Bm + B$	$4Bm + B$	$2B$
Section 3, $\mathbb{Z}_{2^k}$	$5Bm + Bs - 3B$	$4Bm - 3B$	B
QS-Circuit, $\mathbb{Z}_p$	$4Bm + B$	$2Bm + B$	$2B$
QS-Circuit, $\mathbb{Z}_{2^k}$	$3Bm + Bs - B$	$2Bm - B$	B
QS-Poly, $\mathbb{Z}_p$	$3Bm + 2B$	$Bm + 2B$	$2B$
QS-Poly, $\mathbb{Z}_{2^k}$	$2Bm + Bs$	Bm	B

estimate these costs, we used [31] for verifying AND gates at a cost of 7 bits per gate, and [4] for verifying triples in a larger field.

Since COTs and VOLEs can be obtained from pseudorandom correlation generators with very little communication [31, 34], the remaining online communication dominates. Hence, our optimized protocol from Section 3 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 can be optimized by verifying multiplications with QuickSilver [33] or the amortized version of Mac'n'Cheese [4]. This would bring the basic costs closer to our optimized protocols, with the downside of making non-black-box use of information-theoretic MACs, with QuickSilver, or more rounds of interaction and computation, with Mac'n'Cheese. There are two variants of QuickSilver which prove satisfiability of circuits ("QS-Circuit") or of sets of polynomials ("QS-Poly") respectively. Since bitADDcarry can be either represented as a Boolean circuit or as a set of polynomials over $\mathbb{F}_2$, we can use both variants in our conversion protocol.

These results highlight the advantage of our approach compared to using only `daBits`. We also see that using QuickSilver or Mac'n'Cheese to check multiplications (as also done in the concurrent work Mystique [32]) reduces communication by 1.5–3x. This is because verifying a circuit with these protocols is cheaper than evaluating a circuit in our approach (even using faulty triples).

6.2 Experiments

We have implemented the conversion protocol Π_{Conv} using the faulty-dabit approach of Section 3.4 in the Rust programming language using the *Swanky* library[1]. The VOLE protocol over the 61-bit field $\mathbb{F}_p$ with $p = 2^{61} - 1$ is instantiated using [31]. All benchmarks were run on a MacBook Pro 2018, 2.9 GHz 6-Core Intel Core i9, 32 GB 2400 MHz DDR4, with one thread per party. All experiments are run in a Docker container running Ubuntu and using `tc` to artificially limit the network bandwidth, and simulating 1 ms latency.

Table 3: Comm. in Mbit **when verifying 2^{20} conversion tuples where $B = C = 3$ with multiplication check of [31].**

	$m = 8$		$m = 16$		$m = 32$	
	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$
Init	171.8	19.7	174.0	24.2	178.4	33.1
Conv	1488.7	57.8	2522.1	111.1	4591.1	222.2

Table 4: Run-time in s **for verifying 2^{20} conversion tuples with $m = 32$ and $B = C = 3$ with multiplication check of [31].**

Bandwidth	20 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s
Init	22.6	14.0	12.2	12.1
Conv	399.2	189.3	173.5	169.6

Our implementation is currently capable of verifying conversions for $m \leq 60$. We therefore run our conversion protocol on bit lengths $m \in \{8, 16, 32, 60\}$. All benchmarks are run ensuring statistical security of 2^{40}, by varying the sizes of B and C according to table 6 (see Appendix F). Lastly, all of these listed benchmarks are run using Wolverine [31] to verify multiplications as in protocol Π_{Conv} of Section 3.2. We also implemented the variant of our protocol using QuickSilver [33], which reduces the runtimes by around a factor of two, thanks to the lower communication and preprocessing requirements. In Appendix F we list all results when running with either our optimized protocol, or the variant using [33].

Table 3 shows the communication required between the prover and verifier when verifying 2^{20} conversion tuples, when varying the bit lengths.In this table we use Init to define the construction of the channels used by the two parties as well as the initial setup of the Wolverine VOLE protocol and the initial commitments to the provers input. The row Conv covers the time it takes for the prover and verifier to run Π_{Conv} on the input provided by the prover. This covers generation of the required `edaBits`, `daBits` and multiplication triples. Here, even for the smallest setting of $m = 8$ it can already be seen that the conversion costs dominates both the VOLE setup and the Init phase. In Table 4 we list the time for the same setup except the bit length is fixed at $m = 32$. It can be seen that increasing the network bandwidth beyond 100 Mbit/s does not improve the protocol runtime by much. Therefore, for our current implementation, computation is the limiting factor.

Acknowledgements

This work is 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).

[1] https://github.com/GaloisInc/swanky

REFERENCES

[1] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. 2017. Secure Arithmetic Computation with Constant Computational Overhead. In *CRYPTO 2017, Part I (LNCS)*. Springer, Heidelberg.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS 2013*. ACM Press.

[3] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. 2021. Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and $\mathbb{Z}_{2^k}$ (Full Version). Cryptology ePrint Archive, Report 2021/750. https://eprint.iacr.org/2021/750.

[4] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. 41st Annual International Cryptology Conference (CRYPTO 2021).

[5] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91 (LNCS)*. Springer, Heidelberg.

[6] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT 2011 (LNCS)*. Springer, Heidelberg.

[7] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. 2018. Compressing Vector OLE. In *ACM CCS 2018*. ACM Press.

[8] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In *ACM CCS 2019*. ACM Press.

[9] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2019. Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In *CRYPTO 2019, Part III (LNCS)*. Springer, Heidelberg.

[10] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *ACM CCS 2019*. ACM Press.

[11] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN 10 (LNCS)*. Springer, Heidelberg.

[12] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO 2018, Part II (LNCS)*. Springer, Heidelberg.

[13] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2017. The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited. In *CRYPTO 2017, Part I (LNCS)*. Springer, Heidelberg.

[14] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012 (LNCS)*. Springer, Heidelberg.

[15] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[16] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO 2020, Part II (LNCS)*. Springer, Heidelberg.

[17] Shimon Even, Oded Goldreich, and Abraham Lempel. 1982. A Randomized Protocol for Signing Contracts. In *CRYPTO'82*. Plenum Press, New York, USA.

[18] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. 2015. Privacy-Free Garbled Circuits with Applications to Efficient Zero-Knowledge. In *EUROCRYPT 2015, Part II (LNCS)*. Springer, Heidelberg.

[19] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *17th ACM STOC*. ACM Press.

[20] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. 2017. Low Cost Constant Round MPC Combining BMR and Oblivious Transfer. In *ASIACRYPT 2017, Part I (LNCS)*. Springer, Heidelberg.

[21] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT 2020, Part III (LNCS)*. Springer, Heidelberg.

[22] Russell Impagliazzo and Steven Rudich. 1989. Limits on the Provable Consequences of One-Way Permutations. In *21st ACM STOC*. ACM Press.

[23] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO 2003 (LNCS)*. Springer, Heidelberg.

[24] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. 2009. Secure Arithmetic Computation with No Honest Majority. In *TCC 2009 (LNCS)*. Springer, Heidelberg.

[25] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*. ACM Press.

[26] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. 2021. Rabbit: Efficient Comparison for Secure Multi-Party Computation. Financial Crypto 2021.

[27] Moni Naor and Benny Pinkas. 1999. Oblivious Transfer and Polynomial Evaluation. In *31st ACM STOC*. ACM Press.

[28] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *INDOCRYPT 2019 (LNCS)*. Springer, Heidelberg.

[29] Peter Scholl. 2018. Extending Oblivious Transfer with Low Communication via Key-Homomorphic PRFs. In *PKC 2018, Part I (LNCS)*. Springer, Heidelberg.

[30] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. 2019. Distributed Vector-OLE: Improved Constructions and Implementation. In *ACM CCS 2019*. ACM Press.

[31] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. 42nd IEEE Symposium on Security and Privacy (Oakland 2021).

[32] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 501–518.

[33] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. 28th ACM Conference on Computer and Communications Security (CCS 2021).

[34] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast Extension for Correlated OT with Small Communication. In *ACM CCS 2020*. ACM Press.

[35] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT 2015, Part II (LNCS)*. Springer, Heidelberg.

A COMMITMENT FUNCTIONALITIES

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

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

Random On input (Random, id) from $\mathcal{P}, \mathcal{V}$ and where $(\text{id}, \cdot) \notin \text{st}$:
(1) If $\mathcal{P}$ is corrupted, obtain $x_{\text{id}} \in R$ from $\mathcal{S}$. Otherwise sample $x_{\text{id}} \in_R R$ uniformly at random.
(2) Set $\text{st} \leftarrow \text{st} \cup \{(\text{id}, x_{\text{id}})\}$ and send x_{id} to $\mathcal{P}$.
We use the shorthand $[x] \leftarrow \text{Random}()$.

Affine Combination On input $(\text{Affine}, \text{id}_o, \text{id}_1, \ldots, \text{id}_n, \alpha_0, \ldots, \alpha_n)$ from $\mathcal{P}, \mathcal{V}$ where $(\text{id}_i, x_{\text{id}_i}) \in \text{st}$ for $i = 1, \ldots, n$ and $(\text{id}_o, \cdot) \notin \text{st}$:
(1) Set $x_{\text{id}_o} \leftarrow \alpha_0 + \sum_{i=1}^{n} \alpha_i \cdot x_{\text{id}_i}$ and $\text{st} \leftarrow \text{st} \cup \{(\text{id}_o, x_{\text{id}_o})\}$.
We use shorthands such as $[z] \leftarrow a \cdot [x] + [y] + b$.

CheckZero On input $(\text{CheckZero}, \text{id}_1, \ldots, \text{id}_n)$ from $\mathcal{P}, \mathcal{V}$ and where $(\text{id}_i, x_{\text{id}_i}) \in \text{st}$ for $i = 1, \ldots, n$:
(1) If $x_{\text{id}_1} = \cdots = x_{\text{id}_n} = 0$, then send (success) to $\mathcal{V}$, otherwise send (abort) to all parties and terminate.
We use the shorthand $\text{CheckZero}([x_1], \ldots, [x_n])$.

Input On inputs $(\text{Input}, \text{id}, x)$ from $\mathcal{P}$ and $(\text{Input}, \text{id})$ from $\mathcal{V}$ and where $(\text{id}, \cdot) \notin \text{st}$:
(1) Set $\text{st} \leftarrow \text{st} \cup \{(\text{id}, x)\}$.
We use the shorthand $[x] \leftarrow \text{Input}(x)$.

Open On input $(\text{Open}, \text{id}_1, \ldots, \text{id}_n)$ from $\mathcal{P}, \mathcal{V}$ where $(\text{id}_i, x_{\text{id}_i}) \in \text{st}$ for $i = 1, \ldots, n$:
(1) Send $x_{\text{id}_1}, \ldots, x_{\text{id}_n}$, to $\mathcal{V}$.
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 $\mathcal{P}$ sends x to $\mathcal{V}$ and they add $[x] - x$ to the list lst.

MultiplicationCheck Upon $\mathcal{P}$ & $\mathcal{V}$ inputting $(\text{CheckMult}, (\text{id}_{x,i}, \text{id}_{y,i}, \text{id}_{z,i})_{i=1}^{n})$ where $(\text{id}_{x,i}, x_i), (\text{id}_{y,i}, y_i), (\text{id}_{z,i}, z_i) \in \text{st}$ for $i = 1, \ldots, n$:
(1) Send (success) to $\mathcal{V}$ 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 15: Functionality modeling homomorphic commitments of values in the ring R.

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

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

Figure 16: Ideal functionality modeling communication using commitments over multiple domains.

B PROOF OF CORRECT TRUNCATION

Theorem B.1. *The protocol $\Pi_{\text{VerifyTrunc}}$ (Figure 10) UC-realizes $\mathcal{F}_{\text{VerifyTrunc}}$ (Figure 7) in the $\mathcal{F}_{\text{CheckLength}}$-hybrid model.*

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

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

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

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

We now proceed with the proof.

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

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

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

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

proof for the case of a malicious prover.

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

(1) $\mathcal{S}$ forwards the call (VerifyTrunc, N, $\{m^j, [a^j]_M, [a^j_{tr}]_M\}_{j\in[N]}$). If $\mathcal{F}_{\text{VerifyTrunc}}$ returns $\perp$, output $\perp$ to $\mathcal{V}^*$ and abort.
(2) For each $j \in [N]$ $\mathcal{S}$ commits to a random value $[a']_M$ using Input of $\mathcal{F}_{\text{CheckLength}}$. We assume that simulated commitments to $a^j, a'^{,j}$ already exist in $\mathcal{F}_{\text{CheckLength}}$.
(3) For each iteration $j \in [N]$, let l be the size of a^j and m be the size of $a'^{,j}$. $\mathcal{S}$ runs (CheckLength, $\text{id}^{a'^{,j}}, m$) and (CheckLength, $\text{id}^{a^j_{tr}}, l-m$) in $\mathcal{F}_{\text{CheckLength}}$ towards the verifier.
(4) $\mathcal{S}$ then computes $y^j \leftarrow a^j - (2^m \cdot a^j_{tr} + a'^{,j})$ using $\mathcal{F}_{\text{CheckLength}}$ and then runs (CheckZero, id^{y^j}), which it makes output (success).

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

C A NAÏVE TRUNCATION PROTOCOL

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

Protocol $\Pi_{\text{NaiveTrunc}}$

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

Output $[a_{tr}]_M$.

Figure 17: Protocol that naïvely truncates a by m bits

This protocol is much more expensive than our edaBit-based approach, due to working in $\mathbb{Z}_M$ for all operations. Each bit must be committed to by a commitment over $\mathbb{Z}_M$, which itself requires $\log_2(M)$ bits of communication. Furthermore, the checking of each $[a_i]_M$ for $i \in [l]$ requires a multiplication, leading to further interaction. To give an example, we analyze the cost of this protocol when using Wolverine [31] to check the multiplications (alternative protocols such as [4] 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 CheckZero. All calls to 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 $a_i(1-a_i) \stackrel{?}{=} 0$ 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.

A breakdown of the costs of $\Pi_{\text{NaiveTrunc}}$ compared to those of our optimized protocol $\Pi_{\text{VerifyTrunc}}$ (Figure 10) is given in Table 5, where we list both if $\log(M) \geq s$ but also $\log(M) > s$. In both cases, for typical parameters (e.g. $l = 32 \approx \log M$ and $B = 3$–5) the naive protocol has much higher communication cost than ours, since the number of $\mathbb{Z}_M$ openings scales with the bit-length l. To give a concrete number, e.g. for the $\mathbb{Z}_p$ variant with $l = 32 \approx \log M$, when verifying a batch of around a million multiplications and 40-bit statistical security, we can use a bucket size $B = 3$. This leads to the communication of 8256 bits when using the naïve compared to only 960 when using ours, when we disregard the construction of the random authentications in $\mathbb{Z}_2$ and $\mathbb{Z}_p$ for both protocols.

D SUB-PROTOCOLS

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

D.1 Complexity of bitADDcarry

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

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, \ldots, m-1\} \quad (2)$$

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

$$z_i = x_i \oplus y_i \oplus c_i, \forall i \in \{0, \ldots, m-1\} \quad (3)$$

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.

D.2 Complexity of convertBit2A

We consider the procedure convertBit2A as defined in Figure 4. We assume that the input (not the daBit) is distributed prior to running the protocol. This sub-protocol requires a single daBit to

Table 5: Comparison of the costs of $\Pi_{\mathsf{NaiveTrunc}}$ (Figure 17) and $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 10).

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

convert the bit authenticated in $\mathbb{F}_2$ to $\mathbb{F}_M$. Having a single `daBit` $([r]_2, [r]_M)$, we can convert a value $[x_m]_2$ by following the following protocol.

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

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

E PROOFS OF THE $\mathbb{Z}_{2^k}$ PROTOCOLS

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

E.1 Proof of Theorem 5.1

Proof of Theorem 5.1. To show security in the UC-model, we construct a simulator $\mathcal{S}$ with access to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. The environment can choose to corrupt one of the parties, whereupon $\mathcal{S}$ simulates the interaction for the corrupted party. We cover the two cases separately, and first consider a corrupted prover, then a corrupted verifier.

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

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

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

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

For CheckZero, $\mathcal{S}$ first simulates the calls to Random, and runs the protocol with the simulated parties. Then it sends the CheckZero message to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If the simulated verifier aborts, then $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$, which results in the ideal verifier aborting. To show that the verifier's output is indistinguishable between the real execution and the simulation we combine the following two facts: 1. If the verifier aborts in the real execution, then it does the same in the simulation. This holds by definition of the simulation. 2. If the verifier outputs (success) in the real execution, then it does the same in the simulation except with probability at most $\varepsilon_{\mathsf{cz}}$ (defined in Lemma 5.2). We show the contraposition, i.e. if the verifier aborts in the simulation, then it does the same in the real execution except with the given probability. By definition of $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$, the premise hold if one of the input commitments contains a non-zero value. Thus, we can apply Lemma 5.2, which gives us the desired consequence.

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

Since Open is implemented in terms of Affine and 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 5.3, and get that the output of $\mathcal{V}$ is the same in the simulation and in the real execution except with probability at most $\varepsilon_{\mathsf{cz}} + \varepsilon_{\mathsf{cm}}$.

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

advantage of

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

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

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

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

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

For CheckZero, $\mathcal{S}$ sends the respective message to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$, which finishes the simulation. Otherwise, $\mathcal{S}$ simulates the normal protocol execution: It first simulates the calls to Random. Since $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ did not abort, we know that $x_1 = \cdots = x_n = 0$. We also know $\Delta, K[x_1], \ldots, K[x_n], K[r_1], \ldots, K[r_n]$, so we can sample $p_1, \ldots, p_n \in_R \mathbb{Z}_{2^s}$ and compute $M[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + K[x_i] + 2^k \cdot K[r_i]$ for $i = 1, \ldots, n$. Then, the p_i and $h := H(M[y_1]', \ldots, M[y_n]')$ are as expected by the verifier.

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

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

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

E.2 Proof of Lemma 5.2

Proof of Lemma 5.2. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the protocol on commitments $[x_1], \ldots, [x_n]$, but $(x_1, \ldots, x_n) \not\equiv_k (0, \ldots, 0)$. Hence, there is an index $i^* \in \{1, \ldots, n\}$ with $x_{i^*} \not\equiv_k 0$. Thus, also $y_{i^*} \not\equiv_k 0$. Write $\tilde{y}_{i^*} = 2^k \cdot p_{i^*} + \delta$ with $\delta \in \mathbb{Z}_{2^k} \setminus \{0\}$. Let $p_i' \in \mathbb{Z}_{2^s}$ for $i = 1, \ldots, n$ denote the values sent by the prover instead of $p_1, \ldots, p_n$ and define $a := (\Delta \cdot 2^k \cdot p_1' + K[y_1], \ldots, \Delta \cdot 2^k \cdot p_n' + K[y_n])$. Let $h = H(a)$ be the message an honest prover would send, and $h' \in \{0,1\}^s$ the message that $\mathcal{P}^*$ actually sends. $\mathcal{V}$ outputs (success), if $h' = h$ holds. We make a case distinction on how $\mathcal{P}^*$ could produce such an h':

(1) First, $\mathcal{P}^*$ could try to compute the message a that $\mathcal{V}$ inputs into H, and then compute $h' := H(a) = h$. To this end, given $M[y_{i^*}] = \Delta \cdot \tilde{y}_{i^*} + K[y_{i^*}]$, $\mathcal{P}^*$ needs to come up with a value $M[y_{i^*}]' = \Delta \cdot 2^k \cdot p_{i^*}' + K[y_{i^*}]$. Let $v \in \mathbb{N}$ maximal such that $2^v \mid \delta$. By computing

$$\Delta = \frac{M[y_{i^*}] - M[y_{i^*}]'}{2^v} \cdot \left(\frac{2^k \cdot (p_{i^*} - p_{i^*}') + \delta}{2^v} \right)^{-1} \pmod{2^{k+s-v}},$$

$\mathcal{P}^*$ could recover $\Delta \in \mathbb{Z}_s$. Hence, this strategy is successful with probability at most 2^{-s}.

(2) If $\mathcal{P}^*$ is not able to compute a, then $h = H(a)$ is uniformly random from $\mathcal{P}^*$'s view. So whatever message h' it sends, $h = h'$ holds with probability at most 2^{-s}.

By the union bound, $\mathcal{P}^*$ can produce such an h' with probability at most 2^{-s+1}. □

E.3 Proof of Lemma 5.3

Proof of Lemma 5.3. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ 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 [5]. So the verifier outputs (failure).

Therefore, $\mathcal{P}^*$ has two possible options: 1. It can try to cheat during the CheckZero in Step 9 to reveal some different values $d', e' \neq d, e$ or $w_k \neq 0$ in Step 8. This succeeds with probability at most $\varepsilon_{\mathsf{cz}}$ (see Lemma 5.2). 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. [31] have formalized this as a "balls and bins game". According to Lemma 2 of [31], an adversary wins this game with probability at most $\varepsilon_{\mathsf{cm}} = \binom{nB+C}{B}^{-1}$.

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

E.4 Proof of Theorem 5.4

Proof of Theorem 5.4. Since most of $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ is actually identical to $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ we will refer to the Proof of Theorem 5.1 for these parts, and focus on the differences here.

The subroutines CheckZero and CheckZero′ are only very slightly modified from the CheckZero from $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$. The latter is exactly the same as in before, but for the larger message space $\mathbb{Z}_{2^{k+s}}$, and

the former additionally hides some more bits. Hence, the same Lemma 5.2 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 5.1, i.e. $\mathcal{S}$ sends (corrupted, $\mathcal{P}$) to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$ and simulates copies of prover and verifier.

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

If the simulated verifier aborts, it sends (abort) to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. 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 5.5 it also aborts in the real protocol, except with probability $\varepsilon_{\mathsf{cz}} + \varepsilon'_{\mathsf{cm}}$.

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

For CheckMult, we use the same strategy as in the Proof of Theorem 5.1: $\mathcal{S}$ sends the corresponding message on behalf of the corrupted verifier to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$. Otherwise $\mathcal{S}$ simulates the complete protocol using the constant value 0 for all of the prover's commitments 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. □

E.5 Proof of Lemma 5.5

Proof of Lemma 5.5. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the CheckMult protocol with inputs as described in the lemma. Since CheckZero′ is a variant of CheckZero from $\Pi_{\mathsf{ComZK\text{-}a}}^{\mathbb{Z}_{2^k}}$ for the larger message space $\mathbb{Z}_{2^{k+s}}$, we can apply Lemma 5.2 again: Hence, a $\mathcal{P}^*$ that tries to cheat during CheckZero′ is detected by $\mathcal{V}$ except with probability $\varepsilon_{\mathsf{cz}}$.

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

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

Table 6: Conversion tuples that must be checked by Π_{Conv} to ensure statistical security 2^{-s} and bucket size $B = C$.

s	B	# of conversion tuples
40	3	≥1 048 576
40	4	≥ 10 322
40	5	≥ 1024
80	5	≥1 048 576

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:

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

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

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

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

F EXPERIMENTAL RESULTS

We benchmarked our conversion protocol Π_{Conv} from Section 3, as well as a variant which uses Quicksilver [33] to verify the multiplications in bitADDcarry (instead of faulty multiplication triples). We run Π_{Conv} to verify $N = 1024,\ 10\,322,\ 1\,048\,576$ conversion tuples yielding bucket sizes of $B = C = 5,\ 4,\ 3$ respectively, and measure the run-time with different network network bandwiths (20 Mbit/s, 50 Mbit/s, 100 Mbit/s, 500 Mbit/s, and 1 Gbit/s).

Tables 7 and 8 show the measured communication and runtimes for our main protocol. Little to no difference is generally seen between 500 Mbit/s and 1 Gbit/s, showing that the protocol has a bottleneck regarding local computation (however tiny this may be).

Tables 9 and 10 show the measured communication and runtimes for the variant with QuickSilver [33]. Compared to our main protocol, we see a reduction in not only communication, but also the overall running time of the protocol, as both are roughly cut in half. We estimate that this overall gain in efficiency comes from no longer requiring multiplication triples to verify the multiplications, leading to reduced communication and fewer preprocessed COTs. Even if the multiplication triples may be faulty such that for a triple (x, y, z) it may not be true that $x \cdot y = z$, they still require additional communication and COTs when used to verify the bitADDcarry circuits, compared with QuickSilver.

Table 7: The data transferred in Mbit by the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ when verifying N conversion tuples of bit size m with bucket size $B = C$, using our protocol from Section 3.

	$m = 8$		$m = 16$		$m = 32$		$m = 60$	
	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$
$N = 1024, B = C = 5$								
Init	104.65	19.65	104.65	19.65	104.65	19.65	104.65	19.65
Conv	2.45	0.10	4.09	0.01	7.38	0.02	13.13	0.04
$N = 10\,322, B = C = 4$								
Init	105.24	19.65	105.24	19.65	105.24	19.65	105.24	19.65
Conv	19.24	0.10	32.46	0.01	58.89	0.02	107.37	4.48
$N = 1\,048\,576, B = C = 3$								
Init	171.69	19.65	173.92	24.09	178.38	32.98		
Conv	1488.70	57.77	2522.08	111.09	4591.06	222.17		

Table 8: Run-time in s when verifying N conversion tuples of bit size m with bucket size $B = C$ using our protocol from Section 3.

m		20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s
$N = 1024, B = C = 5$						
8	Init	13.6	9.7	8.5	7.3	7.4
	Conv	0.6	0.6	0.7	0.5	0.6
16	Init	13.6	10.3	8.4	7.4	7.3
	Conv	1.2	1.1	1.0	0.9	0.7
32	Init	13.6	9.8	8.5	7.3	7.3
	Conv	2.0	2.0	2.3	1.9	2.3
60	Init	13.6	9.8	8.5	7.4	7.4
	Conv	4.2	3.0	2.9	4.1	2.2
$N = 10\,322, B = C = 4$						
8	Init	13.8	9.8	8.5	7.4	7.4
	Conv	1.3	0.8	0.8	0.8	0.8
16	Init	13.7	9.7	8.6	7.4	7.4
	Conv	2.5	1.6	1.4	1.4	0.1
32	Init	13.7	9.8	8.6	7.4	7.4
	Conv	4.9	2.3	2.6	2.7	2.8
60	Init	13.7	9.7	8.5	7.5	7.4
	Conv	9.8	7.3	6.5	6.2	5.5
$N = 1\,048\,576, B = C = 3$						
8	Init	17.4	11.5	9.6	7.8	7.8
	Conv	120.3	68.8	52.7	46.6	46.0
16	Init	19.2	13.7	11.0	9.3	9.2
	Conv	209.8	123.6	98.5	88.3	87.6
32	Init	22.6	16.4	14.0	12.2	12.1
	Conv	399.2	241.0	189.3	173.5	169.6

Table 9: The data transferred in Mbit by the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ when verifying N conversion tuples of bit size m with bucket size $B = C$ using QuickSilver [33] to verify multiplications.

	$m = 8$		$m = 16$		$m = 32$		$m = 60$	
	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$
$N = 1024, B = C = 5$								
Init	104.65	19.65	104.65	19.65	104.65	19.65	104.65	19.65
Conv	1.46	0.00	2.12	0.00	3.43	0.00	5.72	0.00
$N = 10\,322, B = C = 4$								
Init	105.24	19.65	105.24	19.65	105.24	19.65	105.24	19.65
Conv	11.31	0.00	16.60	0.00	27.17	0.00	45.67	0.00
$N = 1\,048\,576, B = C = 3$								
Init	171.69	19.65	173.92	24.09	178.38	32.98		
Conv	869.12	26.66	1282.92	48.88	2108.28	88.86		

Table 10: Run-time in s when verifying N conversion tuples of bit size m with bucket size $B = C$ using QuickSilver [33] to verify multiplications.

m		20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s
$N = 1024, B = C = 5$						
8	Init	13.6	9.8	8.4	7.4	7.3
	Conv	0.1	0.1	0.1	0.1	0.1
16	Init	13.8	9.7	8.3	7.4	7.3
	Conv	0.1	0.2	0.1	0.1	0.2
32	Init	13.6	9.8	8.4	7.3	7.3
	Conv	0.2	0.1	0.3	0.2	0.2
60	Init	13.5	9.7	8.3	7.3	7.4
	Conv	0.3	0.2	0.1	0.1	0.2
$N = 10\,322, B = C = 4$						
8	Init	13.6	9.7	8.4	7.3	7.3
	Conv	0.6	0.4	0.3	0.3	0.3
16	Init	13.7	9.7	8.5	7.4	7.3
	Conv	0.9	0.5	0.4	0.4	0.4
32	Init	13.6	9.7	8.4	7.5	7.4
	Conv	1.5	0.8	0.6	0.6	0.6
60	Init	13.6	9.7	8.5	7.4	7.4
	Conv	2.5	1.3	1.0	1.0	0.9
$N = 1\,048\,576, B = C = 3$						
8	Init	17.4	11.4	9.4	7.8	7.7
	Conv	66.6	39.9	30.1	24.8	24.8
16	Init	19.2	13.2	10.9	9.2	9.2
	Conv	105.2	65.3	51.2	44.0	43.4
32	Init	22.5	16.3	13.8	12.1	12.1
	Conv	180.6	114.4	93.5	81.6	80.0