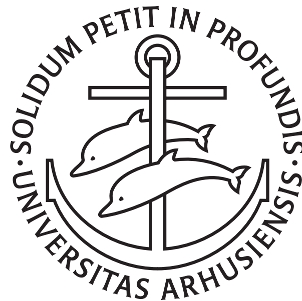
Quattro Formaggi: Zero-Knowledge from VOLE

Alexander Munch-Hansen

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

Quattro Formaggi: Zero-Knowledge from VOLE

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Alexander Munch-Hansen
August 31, 2023

Abstract

Zero-knowledge (ZK) proofs have recently attracted much attention. Typically, these are defined with respect to statements that are fomulated as circuits over a fixed finite field $\mathbb{F}_2$ or $\mathbb{F}_p$ for a large prime p and they are evaluated gate-by-gate. This causes the running time of these protocols to scale linearly with the number of gates. A third domain they can be defined over is the ring $\mathbb{Z}_{2^k}$. These fit what CPUs operate on, thus allows for easier porting of programs to ZK protocols, but little research has been done in ZK over this domain. This thesis pushes forward the study of circuit-based ZK protocols in two major ways; by providing gadgets that lower the overall gate-count and by building new efficient protocols for circuits defined over the ring $\mathbb{Z}_{2^k}$. In addition to this, this thesis also provides a new result in the random-access-memory (RAM) model where we provide a novel ZK proof system. The RAM model allows a prover to evaluate a function represented as a RAM program while revealing no information of the private input. This uses a different computational model (random-access-machines) compared to circuits. Certain programs run much faster in this model versus when represented as a circuit.

Our first contribution presents a maliciously secure, VOLE extension protocol that can turn a short seed-VOLE over $\mathbb{Z}_{2^k}$ into a much longer, pseudorandom VOLE over the same ring. We use these to build additively homomorphic commitments over $\mathbb{Z}_{2^k}$. Moreover, we show that the approach taken by the QuickSilver ZK proof system (CCS 2021) can be generalized to support computations over $\mathbb{Z}_{2^k}$. This new VOLE-based proof system, which we call QuarkSilver, yields better efficiency than previous ZK protocols over $\mathbb{Z}_{2^k}$, at 1.3 million 64 bit multiplications per second in zero-knowledge.

Our second contribution is twofold; two efficient ZK protocols over $\mathbb{Z}_{2^k}$ based on VOLE and a new efficient check of consistency between secret values from the domains of $\mathbb{F}_p$ (or $\mathbb{Z}_{2^k}$) and $\mathbb{F}_2$. This allows the prover and verifier to swap domains of the circuit based on which operations are performed, thus avoiding computing functions in an inefficient domain.

Our third contribution is three new protocols for efficiently evaluating range proofs. These are specialized gadgets for proving that the prover knows some x such that $a \leq x \leq b$ for some public pair (a, b). Each of the provided protocols is efficient in its own right. These are based on either square decomposition or n-ary decomposition, making them better at handling larger or smaller values as well as larger or smaller batches of range proofs.

Our final contribution is a novel ZK proof system for RAM programs compatible with VOLE-based ZK systems such as Quicksilver and Mac'n'Cheese (CRYPTO 2021) that (1) supports arbitrary fields and (2) has linear overhead in both the word size and circuit complexity. This new ZK proof system achieves similar asymptotics are other works, but this approach requires significantly less computation and communication in settings where the number of RAM operations is larger than the RAM.

Resumé

Zero-knowledge (ZK) beviser har i den seneste tid tiltrukket meget opmærksomhed. De er typisk defineret ud fra kredsløb med operationer fra legemerne $\mathbb{F}_2$ eller $\mathbb{F}_p$, hvor p er et stort primtal. Portene i kredsløbet evalueres én af gangen. Dette medfører, at kørselstiden af denne slags protokoller skalerer lineært med antallet af porte. Udover $\mathbb{F}_p$ og $\mathbb{F}_2$ kan man også betragte operationer fra ringen $\mathbb{Z}_{2^k}$. Denne ring passer bedre med, hvordan CPU'er fungerer, hvilket betyder, at normale programmer nemmere kan overføres til ZK protokoller. Desværre findes der dog ikke meget research for dette domæne. Denne afhandling bidrager til forskningen om kredsløb-baseret ZK protokoller på to måder: den giver konstruktioner, der sænker antallet af porte i kredsløb, og den giver nye protokoller for kredsløb defineret ud fra ringen $\mathbb{Z}_{2^k}$. Udover dette giver denne afhandling også et resultat i random-access-memory (RAM) modellen. Her giver vi et nyt effektivt ZK bevissystem. RAM modellen tillader én, der vil bevise noget i ZK, at evaluere en funktion, der er repræsenteret af et RAM program uden at afsløre noget om personens private input. RAM modellen bruger en anden model til at beregne beviser i stedet for kredsløb. Denne hedder "random-access-machines". Visse programmer kører meget hurtigere, hvis denne model bruges, sammenlignet med hvis man brugte kredsløb.

Vores første bidrag er en sikker VOLE protokol, der fungerer ved at udvide få VOLE fra $\mathbb{Z}_{2^k}$ til en masse pseudo-tilfældige VOLE over den samme ring. Vi bruger disse VOLE til at bygge additivt-homomorfiske commitments i $\mathbb{Z}_{2^k}$. Udover dette viser vi, hvordan QuickSilver ZK bevissystemet (CCS 2021) kan blive generaliseret til at fungere i $\mathbb{Z}_{2^k}$. Dette nye VOLE-baserede bevissystem, som vi kalder QuarkSilver, er mere effektivt end nogen andre ZK protokoller i $\mathbb{Z}_{2^k}$, da det kan evaluere 1.3 millioner 64 bit multiplikationer pr. sekund i ZK.

Vores andet bidrag er todelt: to nye ZK protokoller i $\mathbb{Z}_{2^k}$ baseret på VOLE og et nyt effektivt tjek af, om to hemmelige værdier fra de to domæner $\mathbb{F}_p$ (eller $\mathbb{Z}_{2^k}$) og $\mathbb{F}_2$ repræsenterer den samme værdi. Dette tillader én der ønsker at bevise noget i ZK og en der skal verificere det at løbende skifte mellem hvilket domæne kredsløbet er defineret over, altså, at de slipper for at skulle beregne funktioner i et ikke-favorabelt domæne.

Vores tredje bidrag er tre nye protokoller til at evaluere "range-proofs". Disse er specialiserede protokoller til at bevise, at man kender en værdi x således, at $a \leq x \leq b$ for nogle offentligt kendte værdier a, b. Hver protokol som vi foreslår, er effektiv på sin egen måde. Den første protokol er baseret på at opdele tal i kvadrater, og de to andre bygger på en teknik, der hedder n-opdeling. Dette medfører, at de er bedre i forskellige situationer, enten med større værdier, flere værdier, mindre værdier eller færre.

Vores sidste bidrag er et nyt ZK bevissystem for RAM programmer, der er kompatibelt med VOLE-baserede ZK systemer såsom QuickSilver eller Mac'n'Cheese (CRYPTO 2021). Denne nye tilgang (1) tillader arbitrære legemer og (2) skalerer lineært i kost (effektivt set) i forhold til størrelsen på værdierne i hukommelsen og mængden af RAM operationer. På trods af at denne protokol er asymptotisk lige så effektiv som andre protokoller, kræver denne nye protokol markant færre beregninger og mindre kommunikation, når mængen af RAM operationer er større end hukommelsen.

Acknowledgments

The road leading to me now writing these acknowledgements has not been an entirely straight one. To be honest, before I started my master's, I had no plans on becoming a PhD (or heck I didn't even know if I wanted to take a master's degree). I was however eventually convinced to take a master's in Computer Science, during which I took all of the Cryptography courses. This sparked an interest, at which point Claudio Orlandi told me to do a research project during my third semester, which eventually lead to me writing my master's degree with him and Sophia Yakoubov as my advisors. This finally lead to me getting hired by Peter Scholl and Carsten Baum. I am truly grateful for getting the opportunity back during my master's.

However, this thesis simply wouldn't exist without my wonderful advisors, Carsten and Peter. Throughout the last three years, I have spent a great deal of time being confused about things or misunderstanding topics, but both of you have always made absolutely sure that I knew I always had someone to ask and that (to a certain degree) there were no stupid questions. For that I am forever in debt. In addition to this, not only have they taught me much about cryptography and cheese, but also life (getting told several times to go out with friends rather than sit down and work, was apparently required at times). It has been a truly great time, thank you so much.

I am indebted to each and everyone of my collaborators and co-authors: Carsten Baum, Lennart Braun, Alex Malozemoff, Claudio Orlandi, Benoit Razet, Marc Rosen, Peter Scholl and Sophia Yakoubov. I have thoroughly enjoyed working with all of you, thank you!

I want to thank Galois and in particular Richard Jones (and their HR department) for setting aside the tremendous amount of time it took to figure out how to get me a work VISA for the US. My time in the US taught me a lot about how to practically apply my research and I had a great time at Galois (not only due to their snacks).

I want to thank all of the wonderful people in the Aarhus Crypto Group. The small conversations at the coffee machines, that would usually drag out and take waay too long, the chilling in the couches or the impromptu meetings in the hallway. It has been a great time! Specifically I would like to thank Lennart Braun for our many good hours together (some admittedly better than others), both inside and outside of the university.

I would also like to say thanks to all of my friends at Aarhus University. It would certaintly have been a lot more difficult to get to where I am now, without always having people to whom I could rant about my problems (and oftentimes solve them in the process), chill with in the couches in Regnecentralen or collaborate with during the many exercises during my bachelor's and master's degree.

Finally, I'd like to thank my mom. Without her, this thesis would have a grammatically incorrect mess of a resumé. Unfortunately, she gives no guarantee for the remainder of the thesis.

Alexander Munch-Hansen,
Aarhus, August 31, 2023.

Contents

Abstract i

Resumé iii

Acknowledgments v

I Introduction and Overview 1

1 Introduction 3

1.1 Interactive Proofs . . . 3
1.2 Circuit Based Proofs . . . 9
1.3 Commitments . . . 9
1.4 Oblivious Transfer . . . 10
1.5 Commit and prove . . . 14
1.6 SNARKs . . . 22
1.7 Applications . . . 24
1.8 Overview . . . 26
1.9 Additional Publications . . . 35

II Publications 37

2 Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs Over $\mathbb{Z}_{2^k}$ 39

2.1 Introduction . . . 39
2.2 Preliminaries . . . 44
2.3 Single-Point Vector OLE . . . 49
2.4 Vector OLE Construction . . . 65
2.5 QuarkSilver: QuickSilver Modulo 2^k . . . 70
2.6 Experiments . . . 86

3 Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and $\mathbb{Z}_{2^k}$ 95

3.1 Introduction . . . 95
3.2 Preliminaries . . . 99
3.3 Conversions between $\mathbb{Z}_2$ and $\mathbb{Z}_M$ . . . 103
3.4 Truncation and Integer Comparison . . . 122
3.5 Interactive Proofs over $\mathbb{Z}_{2^k}$ . . . 128
3.6 Evaluation . . . 141

4 Cheddar – Oh, Range Proofs for VOLE-based Zero-Knowledge! 151
4.1 Introduction 151
4.2 Square Decomposition 152
4.3 Approximate Range Proofs 158
4.4 Exact Proofs of Lesser Values 165
4.5 Idea based on Polynomials 166
4.6 Evaluation 167

5 Pecorino: More Efficient Zero-Knowledge for RAM Programs 171
5.1 Introduction 171
5.2 Preliminaries 176
5.3 New Protocol for ZK Proofs in the RAM Model 178
5.4 Asserting Permutations 187
5.5 Implementation and Experiments 189

Part I

Introduction and Overview

Chapter 1

Introduction

1.1 Interactive Proofs

Vanessa wants to buy a recipe for a secret sauce from Poul, of which only a tiny sample exist, one which was made many years ago. The thing is, Vanessa knows that the sample is made from the correct recipe, but how can she be sure that Poul is not simply trying to make a quick buck from her, by telling her the wrong recipe? Hell, Poul might not even *know the correct recipe*. Thus, Vanessa is afraid that Poul is trying to fool her and just run with her money, while Poul is afraid that Vanessa might just take the recipe without paying him for it. This leaves the question of how can Poul convince Vanessa that the recipe is correct, without revealing the recipe beforehand? This is where *interactive proof systems* come into the picture.

The idea of an interactive proof system [GMR85, GMR89] was originally formalized by Goldwasser, Micali and Rackoff. To be rigorous about it, we will consider statements x with respect to an instance of a *language* L. This language L can be interpreted as a set of problem instances to which the answer is "Yes", when considering those aforementioned statements. This is denoted by

$$x \in L,$$

in which case the statement is true with respect to the language and the answer is "yes". We model a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$ as two *probabilistic Turing machines*. We then say that $\mathcal{P}$ and $\mathcal{V}$ run an *interactive protocol* to determine if the statement x is in L. The goal of the protocol is for $\mathcal{P}$ to correctly convince $\mathcal{V}$ that $x \in L$, if indeed $x \in L$. To relate this back to our example with Poul trying to sell a recipe to Vanessa, we (informally) define the language L as the problem of checking equality between the supposed recipe and the known-to-be-correct sample of the secret sauce, and the statement is then represented by the recipe that is input by Poul. The goal of the interactive protocol is then to check that the recipe input by Poul is the same one that resulted in the sample. Now, for these interactive proofs to be of interest to us, they must satisfy the following set of properties.

Completeness: making sure the protocol works when both the prover and verifier behaves correctly. Essentially, if both parties behave according to the protocol specification for some input $x \in L$, then $\mathcal{V}$ should always correctly output 1, indicating that indeed x was in L and $\mathcal{V}$ "is convinced" of this fact. Without completeness, protocols clearly fall apart, as they can't be used to prove that something is correct. If the protocol used by Vanessa and Poul lacked completeness, Poul wouldn't be able to convince Vanessa, even *if* Poul was telling the truth and the recipe is correct.

Soundness: protecting against any dishonest behaviour from the prover. Having a protocol that satisfies *completeness* is naturally a necessity, otherwise what is the point, if the protocol fails despite both parties behaving honestly? However, it is a steep demand that everyone must always behave honestly. Therefore we must have a way of catching $\mathcal{P}$, in case $\mathcal{P}$ behaves dishonestly. Specifically, what happens if the statement x isn't in the language L? To avoid such shenanigans, the protocol should also be *sound*, or rather, satisfy *soundness*. This implies that any potentially malicious or cheating $\mathcal{P}$, henceforth denoted as $\mathcal{P}^*$, will have a hard time convincing $\mathcal{V}$ that $x \in L$, if, in reality, $x \notin L$. If this was not the case, it implies that Poul can get away with the money, without actually giving the correct recipe to the secret sauce, which would leave Vanessa without both recipe *and* money.

Zero-Knowledge: protecting against overly curious verifiers. This curiously named property is perhaps a bit of a misnomer. It refers to the fact that during an interactive proof between a prover and a verifier, the verifier should learn only that $x \in L$ (or $x \notin L$), but nothing more than this. This should be true, regardless of whether or not the verifier behaves honestly. Without this property, Vanessa could simply learn the recipe through the proof that Poul is creating purely so that she can be convinced that he actually knows the recipe! This is obviously not in the best interest of Poul, as he will now have given the recipe and now has to trust Vanessa to give the money, despite her already knowing being able to cook the secret sauce now. Thus, zero-knowledge is clearly an important property.

However, if a verifier has infinite computing power and time, it becomes very difficult to define a protocol that can never be broken, e.g. one where the verifier can never learn the secret from the proof. To remedy this, we usually consider very specific problems to base protocols on, in addition to bounding the time the verifier has to solve the problems.

The protocols usually considered are those that, when one is given a problem as well as a potential solution, one can verify that the solution is correct in an efficient way, but without the solution, a new solution is very difficult to find. The class of such problems is known as NP. To relate this idea back to the discussion on turing machines earlier, it is the set of problems that can be solved in polynomial time by a *nondeterministic* turing machine, or rather, if a problem is given to a verifier who does not know the solution, the verifier would likely have to try all possible combinations of inputs before finding a result. This is a slight simplification as P $\subset$ NP and even if we consider the problems which are NP-complete (i.e. problems that are in NP but not in P), then in some cases there do exist algorithms which are faster than bruteforce. Disregarding this for now, if the verifier would have to try all possible combinations of inputs before finding a result, this makes the computation time grow exponentially in the number of inputs. We call the possible solution to the problem, the *witness* (usually w for short). We can now rephrase the above definition of zero-knowledge. If the verifier is given a proof $(x, w) \in L \in NP$, the verifier should not learn anything about w apart from the fact that $(x, w) \in L$. Likewise, the prover cannot come up with a witness w without knowing

one before-hand, due to the definition of NP. It is worth noting that this definition is a bit too restrictive, as it is known that the class of languages for which there exists interactive proofs for, is, in fact, as large as PSPACE [LFKN90, Sha90], however the interactive proofs for PSPACE are not practical for the prover and will not entertain this further here.

Now, having defined what zero-knowledge means to a protocol, we are left with the task of how to show that a protocol satisfies it. From the above discussion, this is not immediately obvious, as all we've done is conclude that a verifier should learn nothing in regards to the witness.

However, how to exactly define *learn nothing*, turns out to be an intricate task.

1.1.1 Simulation

Simulation is a key to how we try to define the concept of *learning nothing*. Now that we know that any protocol satisfying zero-knowledge must mean that $\mathcal{V}$ learns nothing, then it seems appropriate to define what exactly we mean by this. We mentioned that $\mathcal{V}$ shouldn't learn anything, except that $x \in L$, but what does this mean mathematically? The *simulation paradigm* seeks to answer this question. This paradigm is one of the most important contributions to cryptography and it has been used to define security notions in a wide variety of cryptographic branches, such as zero-knowledge, multi-party computation and public-key cryptography [Gol01, HL10, CDN15, Lin16].

The concept of simulation in this context, is a way of comparing something that happens in a *real* world, to that of an *ideal* world. In this ideal world, we make the assumption that the primitive in question is secure by definition. Specifically, in the *ideal* world, there is no prover and we only consider a potentially malicious verifier $\mathcal{V}^*$ as well as the statement x. We then see a *simulator* Sim as an imaginary entity that is spawned in the ideal world, thus, whatever Sim produces, corresponds to the information that an attacker $\mathcal{V}^*$ would be able to compute on his own in this ideal world (as the Sim is not actually the prover). We can then compare this information to what the verifier $\mathcal{V}^*$ sees in the *real* world, where $\mathcal{V}^*$ actually does run a protocol with a prover $\mathcal{P}$. Finally, we can define that a proof system is (computationally) zero-knowledge, if for any probabilistic polynomial-time malicious verifier $\mathcal{V}^*$ there exists an efficient simulator Sim that can produce a *view* of $\mathcal{V}^*$ (i.e. all the information that $\mathcal{V}^*$ observes as partt of the protocol), that is indistinguishable from the real world. If that is indeed the case, then $\mathcal{V}^*$ must have learned nothing new in the real world, as Sim does not know the secret of $\mathcal{P}$ in the first place.

To this end, another benefit of the simulation paradigm, is that once a protocol in the real world has been shown to be indistinguishable from the ideal world, one can henceforth use the ideal definition as part of other protocols. This simplifies proofs significantly, as we can assume the ideal version is correct by definition. This allows us to prove submodules of protocols secure and replace them with their ideal version in the overall protocol. We are then left with a protocol that neither exists in the real world nor the ideal world. For this, we create a *hybrid world* in which we then prove the protocol secure. This kind of composition is known as the *Universal Composition*

framework or the *UC-model* [Can00].

Having established that a protocol satisfying zero-knowledge means having a protocol based on a problem that is difficult to solve, we will now consider what this actually means.

1.1.2 Dificulty of Problems

What remains is how we actually pick problems or computations that we hope are difficult (unfortunately we cannot prove that the problems are difficult, without also resolving $P \neq NP$, at which point we'd probably have bigger problems).

When creating a new protocol, the authors usually base this on exisiting problems that have shown long-standing resistance to attempts from experts trying to design efficient algorithms solving these problems.

We therefore propose new constructions based on existing well-established assumptions, by showing that the security of our new construction is captured by the security of some well-established assumption. This is what is known as a cryptographic reduction: Let A be some well-established assumption. A proof under this assumption A, is a proof that, if there exist any adversary B that can somehow abuse the construction in polynomial time, then there exist some other adversary B' that can, also in polynomial time, contradict the assumption A. More specifically, these reductions typically work by the authors providing an explicit adversary B' that runs B internally in order to break the assumption A.

The idea of proving security by reduction creates an interesting string of seemingly unrelated primitives that are all used in completely different situations and achieving different goals, but they are connected by their security reduction to some well-established assumptions. This notion of proving security was presented in [GM82].

1.1.3 Brief Introduction to Algebra

We've established (on a high level) what an interactive proof is and how we prove it to be secure. If we then make up a protocol and implement it using our regular numbers (i.e. $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$ ad infinitum), it turns out we'd run into problems. The easiest explanation to this problem, is that we can simply not represent all of these numbers on computers, as we have a finite amount of space. We therefore must construct finite sets of numbers over which we can implement our protocols, but these things must be defined in such a way that any hardness assumptions (the so-called "difficult" problems mentioned in Section 1.1.2) are acrtually hard.

This section contains a very brief introduction to field and ring arithmetics. If the reader is familiar with this, then feel free to skip this. The integers are the usual numbers $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$ ad infinitum, which are represented by the set $\mathbb{Z}$. Addition and multiplications of two elements $a, b \in \mathbb{Z}$ leads to another element $c \in \mathbb{Z}$, meaning the set Z is *closed* under these operations. This also holds true for subtraction, since it can be defined from addition. It is *not* the case for division however. Consider $\frac{5}{2}$ which is 2.5, which is *not* an integer (but instead a *real* or *rational* number).

Rings using the definition of the integer set $\mathbb{Z}$, we define the concept of rings. A ring is a set R which has two binary operations, additions and multiplication, satisfying the following properties:

1. Addition is/has:

   associative meaning that $a + b) + c = a + (b + c)$ for all $a, b, c \in R$

   commutative meaning that $a + b = b + a$ for all $a, b \in R$

   identity meaning that there exists an element b such that $a + b = a$ for all $a \in R$. I.e., $b = 0$

   inverse meaning that for all $a \in R$, there exists some $-a$ so that $a + (-a) = 0$

2. Multiplication is/has:

   associative meaning that $a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$

   identity meaning that there exists an element b such that $a \cdot b = a$ for all $a \in R$. I.e., $b = 1$

   left/right distributivity $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$ for all $a, b, c \in R$

If the multiplicative operator of some ring is commutative as well, then that ring is called a commutative ring. If a ring R, in addition to satisfying the above, also has the following property:

Multiplication Inverse meaning that for all $a \in R$, $a \neq 0$, there exists some a^{-1} called the multiplicative inverse of a so that $a \cdot a^{-1} = 1$

then we call R a *field*.

Interestingly, the set of integers $\mathbb{Z}$ does form a ring (which the avid reader can check for themselves), but it does not form a field, as the only non-zero integers that have multiplicative inverses are -1 and 1.

To salvage this situation, we can equip the set $\mathbb{Z}$with a modulo M, denoted as $\mathbb{Z}/M\mathbb{Z}$ of oftentimes just $\mathbb{Z}_M$. As an example of a ring, consider $R = \mathbb{Z}_{16}$. Now, the set R contains the integers $0, \ldots, 15$ and any operation is followed by a modulo 16. This is still not a field however. For instance, an inverse does not exist for $4 \in \mathbb{Z}_{16}$. If we instead equip Z with a modulo p that happens to be a *prime*, we get what is known as a *prime field* $\mathbb{Z}_p$. An example of such a field is $\mathbb{Z}_{17}$. Here, one can check that for any non-zero value $x \in \mathbb{Z}_{17}$, there exists an inverse such that $x \cdot x^{-1} = 1$. As such, $\mathbb{Z}_{17}$ is a field.

It turns out that prime fields in particular has some very nice properties when creating cryptographic protocols and this field (no pun intended), has seen a lot of research. Less common has it been for rings to get much interest, but as of lately more protocols has started appearing with focus on working over the ring $\mathbb{Z}_{2^k}$ for some integer k [CDE+18, BBMH+21b, GNS21, BBMHS22a].

1.1.4 Computation over $\mathbb{Z}_{2^k}$

An interesting structure that has already been briefly introduced in Section 1.1.3 is the set of integers modulo 2^k, denoted as $\mathbb{Z}_{2^k}$. Most existing research has been focused on the topic of zero-knowledge over fields. This is primarily due to the difficulties of working over rings rather than fields, such as rings having zero-divisors as well as them lacking multiplicative inverses for all elements. There are however several reasons why working in this particular area may be of interest.

Computers work in powers of two Computer hardware architectures typically run using 32-bit or 64-bit integers. This corresponds directly to arithmetics over $\mathbb{Z}_{2^k}$ for $k = 32$ or $k = 64$, respectively. This avoids having to implement the reduction modulo p operation in software. For some choices of p, so-called *Mersenne primes*, this reduction can be implemented fairly cheaply. We postulate however, that with proper implementations, implementing protocols directly for the underlying hardware, i.e. working over a ring $\mathbb{Z}_{2^k}$ rather than a finite field, could lead to an improvement in performance.

Specialized protocols Some protocols and primitives can, particularly well, utilize working over $\mathbb{Z}$. One example of this, is being able to use the algebraic structure of linearly-homomorphic IT-MACs based on VOLE so that one can take multiple commitments $[x_0], ..., [x_{k-1}]$, where $[x_i] \in \mathbb{F}_{2^k}$ and $x_i \in \mathbb{F}_2$, and pack these into a single IT-MAC for $x \in \mathbb{F}_{2^k}$. This procedure is injective. Secondly, some operations such as truncation and integer comparisons are trivialized when working over $\mathbb{Z}_{2^k}$ instead of $\mathbb{F}_p$ for some prime p.

Furthermore, it holds true that

Functions such as truncations and integer comparisons are trivialized when working over $\mathbb{F}_2$.

There are however also some issues with working over rings $\mathbb{Z}_{2^k}$ for any integer k. In $\mathbb{Z}_{16}$ for instance, the polynomial $x^3 - 8$ has 4 roots, $2, 6, 10, 14$. Instead, if we work over the prime field $\mathbb{Z}_{17}$ and consider the same polynomial, $x^3 - 8$, there is only a singular root; 2. This, in practice, is known as the *Schwartz-Zippel lemma* [Zip79, Sch80]. This also has an effect on any protocol utilizing *information-theoretic message authentication codes* (or *IT-MACs*). The security of these relies on equations of the type $\mathsf{X} \cdot \delta + \gamma = 0$ to have only one possible solution for X (provided that $\delta \neq 0$). However, as we have just seen a very similar situation during the discussions on polynomials, this is not the case when considering rings $\mathbb{Z}_{2^k}$, where the above polynomial was defined over $k = 4$ (or rings in general). In the more general case where the ring is simply $\mathbb{Z}_{2^k}$, it turns out that, for the equation $\mathsf{X} \cdot \delta + \gamma = 0$, if we let $\delta = 2^{k-1}$ and $\gamma = 0$, then if we let X be any even number, then $\mathsf{X} \cdot \delta + \gamma = 0$ is true. Thus, working over a ring breaks this type of system. This has implications when we try to design zero-knowledge protocols requiring the prover to commit to certain values and then later open these, as this uses this exact type of equation, it allows the prover to break the *binding* property of the commitment scheme used.

1.2 Circuit Based Proofs

Having introduced the two (or three) primary models of computation; over $\mathbb{Z}_p$, over Z_{2^k} (or $\mathbb{Z}_2$), we are now ready to introduce a way of actually implementing these interactive protocols that was introduced earlier.

Arithmetic circuits are a model of computation [CD98] that differs from the Turing machines mentioned above. In this model, we instead consider circuits having gates and wires, where all (except for one) gates having input wires represent some operation. If a gate has no input wire, it represents a circuit input wire, which is used to supply the input to the circuit. The single wire that originates from a gate but which does not have an output gate, is the output wire. The input wires are then fed with inputs to the computation and processed through the gates in order to obtain values for the internal wires representing the intermediate results achieved from computing the operations of the gates. Finally, the output wire is reached, where the final result of the computation can be obtained from.

The gates in the circuit represent operations over a ring, which allows for the two binary operations addition and multiplication. In this case, each gate is represented by either addition or multiplication. A typical choice of ring is the set of integers modulo a prime p, $\mathbb{Z}_p$ (i.e. a *field*). When this particular type of ring is picked, every non-zero element $x \in \mathbb{Z}_p$ has the extra property of having a multiplicative inverse (or rather, for $x, \in \mathbb{Z}_p$, there exists some $y \in \mathbb{Z}_p$ so that $\frac{x}{y} = 1$). Fields, having this exact property, are more malleable and permit the use of special techniques and constructions that are not possible were it not for this property.

A particular choice of field, is the set $\{0, 1\}$ with the operations AND and XOR. In this case, the modulo used is 2 and arithmetic circuits defined over this field are known as *binary circuits*. These are highly important in many use cases, as particular computations are more easily defined when considering binary representations. Another commonly used field is that of $\mathbb{Z}_p$ for a large prime p.

1.3 Commitments

Making a commitment simply means that a party in a protocol picks some value from a finite set of values and then commit to this choice once and for all, so that the party can no longer change this. The idea is originally due to [BCC88]. As an informal example:

- $\mathcal{P}$ wants to commit to a number x. To do so, $\mathcal{P}$ writes down the number x on a piece of paper, and then puts this piece of paper in a locked box.
- $\mathcal{P}$ gives this box to $\mathcal{V}$.
- $\mathcal{P}$ can now choose to open this commitment by giving the key to $\mathcal{V}$.

This simple example highlights two basic properties of any commitment scheme:

1. When $\mathcal{P}$ gives away the box to $\mathcal{V}$, $\mathcal{P}$ can no longer change the choice of made, as this is stashed inside the locked box. Thus, when/if $\mathcal{P}$ opens the box to $\mathcal{V}$, $\mathcal{V}$ can be sure that it is indeed the number picked initally by $\mathcal{P}$. This is known as the *binding* property.

2. The piece of paper being stashed inside a locked box, means $\mathcal{V}$ can't tell the number, even by having the box. This is known as the *hiding* property.

Apart from these two basic properties, commitment schemes can be what is known as *homomorphic*. Formally, this describes the transformation of data in a structure preserving way, but to give a brief summary, it means that certain operations can be computed on the data that has been committed to, whilst allowing $\mathcal{P}$ to eventually open the commitment correctly. A particular type of homomorphic property is when commitments are *linear* or *additive*. This allows for certain linear operations to be performed on the committed values, such as addition. This turns out to be very useful when considering circuit-based zero-knowledge proofs, as circuits are comprised of addition and multiplication gates.

To give an example, let $\mathbb{Z}_p$ be a finite field defined from a large prime p:

- $\mathcal{P}$ commits to $x, y \in \mathbb{Z}_p$, resulting in $\mathcal{P}$ and $\mathcal{V}$ having $[x], [y]$.
- $\mathcal{P}$ and $\mathcal{V}$ $[z] \leftarrow \mathsf{Add}([x], [y])$
- $\mathcal{P}$ computes $z \leftarrow \mathsf{Open}([z])$.
- Now, $\mathcal{V}$ can trust that $z = x + y$, due to the homomorphic properties of the commitment scheme.

However, if $\mathcal{P}$ and $\mathcal{V}$ has to do any multiplications, it becomes more tricky. This is not supported by this particular commitment scheme, so the two parties will have to do something else. There are however various ways of achieving this, either by using so-called Beaver Triples [Bea92] (an idea we will elaborate on in Section 1.5) or by more complicated means [WYKW21a]. There also exist fully homomorphic commitment schemes. These allow for local computation of both addition and multiplication of hidden values, but instantiating these usually require costly public key cryptography [MSM$^+$22] and for our applications it is not worth it, to instantiate such constructions.

There exists many ways of instantiating a commitment scheme satisfying the above properties, but in this thesis we focus on one in particular. This specific construction is based on *Vector Oblivious Linear-Function Evaluation* or *VOLE*. Before we can introduce this primitive, we must first describe another primitive called *Oblivious Transfer*

1.4 Oblivious Transfer

Oblivious Transfer is not strictly something that is considered in this thesis, but it is used as a tool in some constructions, which is why we give a proper introduction to it. Furthermore, while future sections are derived from oblivious transfer, they can

stand alone without the reader having a thorough understanding of oblivious transfer. Regardless of this, we now introduce oblivious transfer.

Imagine that Bob has two padlocks with no distinguishing features, apart from that fact that Bob has a single key to exactly one of them. Bob then marks the padlocks with a 0 and 1 respectively and lets $c \in \{0,1\}$ represent which padlock he can open. Bob then gives both padlocks to Alice, who writes a message on two pieces of paper, puts these into two boxes, denoted by b_0 and b_1, respectively. Alice then locks the two boxes with their correspondent padlock and gives them back to Bob. Now, Bob can open exactly one box b_c, i.e. the one locked with padlock c, while not learning anything about the other box. Additionally, Alice learns nothing of which box Bob could open.

Formally, this is defined as a $1-2$ oblivious transfer (or OT) protocol [Rab05], is one in which a sender $\mathcal{S}$ knows two bits b_0, b_1 and a receiver $\mathcal{R}$ has a single choise $c \in \{0,1\}$. The protocol allow $\mathcal{R}$ to learn exactly b_c, but nothing about the other bit. Likewise, $\mathcal{S}$ learns nothing about c, i.e., which choice $\mathcal{R}$ made. This simple notion, can be generalised to what is known as $1-n$ OT. Perhaps obvious from the name, this allows $\mathcal{R}$ to input a challenge $c \in \{0,\ldots,n-1\}$ instead and symmetrically $\mathcal{S}$ inputs $b_0,\ldots,b_{n-1}$, but the same aforementioned guarantess holds true for this case as well.

An additional abstraction we can make, is to define *random* OT. In this setting, the sender and receiver instead get $(b_0, b_1$ and c, b_c, respectively. Oftentimes we disregard c when considering random OTs.

Lastly, something which will be required later on, is the term *extension* in the context of Oblivious Transfers [IKNP03] (more commonly called *OT-extension*). Here, it refers to reducing the cost of doing many OTs by working in two phases:

setup in this first phase, a small number of *seed* OTs are produced using standard (relatively expensive) techniques.

extension in this second phase, the seed OTs are *extended* to create many more OTs with a much lower cost than the seed OTs. This second phase is usually based on much cheaper cryptography known as *symmetrical* cryptography.

This extension phase is of particular interest, as it has been proven to be impossible to create OTs using strictly symmetrical cryptography [IR90].

Lastly, we can define *correlated oblivious transfer* (or COT). In this case, the receiver learns $(b, w = v + \Delta \cdot b$ where $\Delta, v \in \mathbb{F}_2$ and the sender learns $(v, v + \Delta)$. Here we say that b is the choice bit of the receiver. An interesting thing to note here is that since we are working in $\mathbb{F}_2$, the receiver either learn v or $v + \Delta$, based on the choice bit $b \in \{0,1\}$.

1.4.1 Oblivious Linear Evaluation

We can then make a generalization of Oblivious Transfer and COTs, which we call *Oblivious Linear Evaluation*. An Oblivious Linear Evaluation (or OLE), is a two-party protocol between a sender and a receiver. The protocol yields a tuple of correlated values (r, m, k, Δ) such that

$$m = k + \Delta \cdot r.$$

This idea seems very close to COTs and in fact it is the same idea, just in a larger field. Formally, when we consider the binary setting, we consider COTs, but this differentiation is not always made, and then the general term OLE will be used in all cases.

The receiver receives (k, Δ) and the sender receives (m, r). Now that we get into the territory of what we need to explain the results of this thesis, we'll formalize this further. The value r obtained by the receiver is defined such that $r \in \mathbb{F}_p$ whereas the remaining values come from the the *extension field* $\mathbb{F}_{p^\ell}$: $\Delta, m, k \in \mathbb{F}_{p^\ell}$. The extension field $\mathbb{F}_{p^\ell}$ is defined via $\mathbb{F}_p$ so that $\mathbb{F}_p \subseteq \mathbb{F}_{p^\ell}$. We define Δ as a global key, which is sampled uniformly once and then kept by the receiver throughout the protocol. For each OLE, $k \in \mathbb{F}_{p^\ell}$ is sampled randomly and likewise is $x \in \mathbb{F}_p$. Due to addition of k, this makes $m \in \mathbb{F}_{p^\ell}$ look random as well.

This can be done in batches (where Δ is fixed across the batches but the values and keys are fresh), at which point this is called *vector-OLE* (or VOLE). We then write

$$\mathsf{M}[r] = \mathsf{K}[r] + \Delta \cdot r \in \mathbb{F}_{p^\ell}.$$

The reason why the introduction of VOLE is necessary, is because we can build *information-theoretic message authentication codes* (IT-MACs) [NNOB11, DPSZ11] from these. In turn, we can build a commitment scheme from these IT-MACs which is linearly homomorphic. This comes with a significant boost in efficiency both in communication as well as computation, as this can be done through extension techniques (something we briefly introduce later), as compared to earlier techniques. To simplify notation, we redefine the sender $\mathcal{S}$ to be the prover $\mathcal{P}$ and the receiver $\mathcal{R}$ to be the verifier $\mathcal{V}$. Now, the prover receives $(r, \mathsf{M}[r])$ and the verifier recieves $(\mathsf{K}[r], \Delta)$. The prover can then commit to any value $x \in \mathbb{F}_p$ of the provers choice, by going through the following steps (which we define as Input):

1. The prover computes $\delta \leftarrow r - x$ and then δ to the verifier.
2. The verifier computes $\mathsf{K}[x] = \mathsf{K}[r] + \delta \cdot \Delta$.
3. Finally, $\mathsf{M}[x] = \mathsf{M}[r]$. Now, the tuple $(x, \mathsf{M}[x])$ held by the prover and the tuple $(\Delta, \mathsf{K}[x])$ held by the verifier, is a commitment to x, henceforth denoted as $[x]$.

The point from Step 3 follows from:

$$\begin{aligned}
\mathsf{M}[x] &= \mathsf{K}[x] + \Delta x \\
&= (\mathsf{K}[r] + \delta \cdot \Delta) + \Delta x \\
&= (\mathsf{K}[r] + (r - x) \cdot \Delta) + \Delta x \\
&= (\mathsf{K}[r] + \Delta r - \Delta x) - \Delta x \\
&= \mathsf{K}[r] + \Delta r \\
&= \mathsf{M}[r]
\end{aligned}$$

which shows that we can turn the new commitment into the old by substituting the provided values.

Despite us having been calling the above a commitment scheme, we have yet to show that this is in fact a secure one. We do this, by showing that the scheme satisfies the *binding* and *hiding* property, that was shown in Section 1.3.

hiding The property of hiding is satisfied from the fact that the verifier obtains no information on the committed-to value x. This follows from the security of the (V)OLE construction used, as this ensures that the verifier obtains no information regarding $(r, \mathsf{M}[r])$. Now, since the verifier has no information about r, the verifier can't deduce anything from seeing $\delta = r - x$.

binding The property of binding is satisfied from the fact that the prover has no information regarding $(\mathsf{K}[r], \Delta)$ (which also follows from the security of the (V)OLE). Specifically, if the prover wishes to change its value x to some x', this requires changing $\mathsf{M}[x]$ to $\mathsf{M}[x'] \leftarrow \mathsf{M}[x] + \Delta(x' - x)$, but, since the prover has no information on Δ, this can only be done by guessing $\Delta \in \mathbb{F}_{p^\ell}$, which happens with probability $\frac{1}{|\mathbb{F}_{p^\ell}|}$, where $|\mathbb{F}_{p^\ell}|$ means the size of the extension field.

We say that the prover and verifier runs the following operation $[x] \leftarrow \mathsf{Input}(x)$ on the prover-known input x, resulting in the commitment $[x]$. We also define the procedure $\mathsf{CheckZero}([x])$ as simply having $\mathcal{P}$ and $\mathcal{V}$ open the value and check that it is 0. Rather than do this on a per-element basis, this procedure can be run in batches of several field elements, but as this procedure is more complicated and not required for our needs, we will not touch on this anymore.

Finally, all we have left to show, is that this type of commitment scheme is, in fact, additively homomorphic. If $\mathcal{P}$ and $\mathcal{V}$ hold two authenticated values $[x]$ and $[y]$, then they can locally compute $[z] = [x + y]$. This is done by having $\mathcal{P}$ compute $z \leftarrow x + y$ as well as $\mathsf{M}[z] \leftarrow \mathsf{M}[x] + \mathsf{M}[y]$. $\mathcal{V}$ locally computes $\mathsf{K}[z] \leftarrow \mathsf{K}[x] + \mathsf{K}[y]$. This becomes apparent once we write out the above equations:

$$\begin{aligned} \mathsf{M}[z] &= \mathsf{K}[z] + \Delta z \\ \mathsf{M}[x] + \mathsf{M}[y] &= (\mathsf{K}[x] + \Delta x) + (\mathsf{K}[y] + \Delta y) \\ &= (\mathsf{K}[x] + \mathsf{K}[y]) + \Delta(x + y) \end{aligned}$$

The above set of operations is denoted as $[z] = [x] + [y]$ for brevity.

Similarly, this holds true for addition as well as multiplication with any public value $y \in \mathbb{F}_p$, which results in $[z] = [x + y]$ or $[z] = [x \cdot y]$. Both of these operations follow the above steps, are these are denoted as $[z] \leftarrow [x] + y$ and $[z] = [x] \cdot y$, respectively.

This does not hold true however, for multiplication of two hidden values. Consider the following:

$$\begin{aligned} [x] \cdot [y] &= (\mathsf{K}[x] + \Delta x) \cdot (\mathsf{K}[y] + \Delta y) \\ &= \Delta^2 xy + \Delta \mathsf{K}[x] y + \Delta \mathsf{K}[y] x + \mathsf{K}[x]\mathsf{K}[y] \end{aligned}$$

which contains terms such as $\Delta^2 xy$. Here, each term contains values known by *either* the prover *or* the verifier, making them not *immediately* locally computable without communication. It is worth noting that if we take a commitment $(x, \mathsf{M}[x])$ held by the prover and $\mathsf{K}[x]$ held by the verifier, then we can view these instead as a polynomial so that $f(y) = \mathsf{M}[x] - x \cdot y$. Now $\mathsf{K}[x] = f(\Delta) = \mathsf{M}[x] - \Delta \cdot x$. If we then multiply the polynomial defined above with another $f'(y) = \mathsf{M}[x'] - y \cdot x'$:

$$f \cdot f'(y) = \mathsf{M}[x]\mathsf{M}[x'] + (x\mathsf{M}[x'] + x'\mathsf{M}[x])y + xx'y^2$$

we see that the final term contains $xx' \cdot y^2$ which is the only term of degree 2. This however, is so far locally computable. The prover then define polynomial $q(y) = \mathsf{M}[z] + z \cdot y$ where the prover claims that $z = x \cdot x'$. This requires committing to z, so the verifier learns $\mathsf{K}[z] = q(\Delta)$. Now, to show that this is true, the prover can send over the polynomial

$$y \cdot q(y) - f \cdot f'(y)$$

where $y \cdot q(y) = y\mathsf{M}[z] + zy^2$. Thus, if $z = x \cdot x'$, the term of degree 2 disappears. On the verifier side of things, the verifier has $mackeyx \cdot \mathsf{K}[x'] = f \cdot f'(\Delta)$ and can compute $\mathsf{K}[x]\mathsf{K}[x'] - \Delta \cdot \mathsf{K}[z]$, which, as on the prover side, removes the final term of highest degree, if the computation is correct. While only working on a single multiplication, this idea is not more efficient than a regular multiplication. Howver, this idea generalizes to some degree d, i.e. the prover and verifier can combine d multiplications. The final remark to make regarding this idea, is that the polynomial that the prover sends over must be masked by a degree $d - 1$ polynomial, to mask the remaining terms.

While the above may have seemed like a detour, it is a fact we will need later on in this thesis. See step 2 in Figure 1.3 for an example of how multiplication can be computed using additively homomorphic commitments in a simpler fashion than what was just described, using multiplication triples.

Finally, we define the ideal functionality $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ (Figure 1.1) that we can use to generate VOLE for our zero-knowledge protocol. A protocol satisfying this functionality over $\mathbb{F}_p$ can be found in [WYKW21a].

This functionality has two steps; an initilization step in which the verifier gets the global key Δ and an extension step which allows the prover and verifier to generate *random* VOLEs. The last thing to notice here is that each step allows for either the prover or the verifier to be malicious, in which case they sample their own values rather than having them be random. This is an important part, as it allows the simulator later on to choose the values, when interacting with either a maliicous prover or a malicious verifier, but it also gives more power to the adversary.

1.5 Commit and prove

This is a way of proving satisfiability of a circuit in zero-knowledge. In this case, the prover commits to each input $([w_1], \ldots, [w_n]) \leftarrow \mathsf{Input}(w_1, \ldots, w_n)$. The prover and verifier then utilise the homomorphic properties of the commitment as well as some

Functionality $\mathcal{F}_{\mathsf{VOLE}}^{l,p}$

Initialize: On receiving Init from $\mathcal{P}$ and $\mathcal{V}$, sample $\Delta \leftarrow \mathbb{F}$ if $\mathcal{V}$ is honest. Otherwise, receive $\Delta \in \mathbb{F}$ from $\mathcal{V}$. Store the global key Δ and send Δ to $\mathcal{V}$. Ignore all subsequent Init commands.

Extend: On receiving (Extend) from $\mathcal{P}$ and $\mathcal{V}$:

1. Sample $r \leftarrow \mathbb{F}_p, \mathsf{K}[r] \leftarrow \mathbb{F}_{p^\ell}$ and compute $\mathsf{M}[r] = \mathsf{K}[r] + \Delta \cdot r \in \mathbb{F}_{p^\ell}$.
   - If $\mathcal{P}$ is corrupted: receive $r, \mathsf{M}[r]$ from $\mathcal{A}$ and let $\mathsf{K}[r] = \mathsf{M}[r] - r \cdot \Delta$.
   - If $\mathcal{V}$ is corrupted: recieve $\mathsf{K}[r]$ from $\mathcal{A}$ and let $\mathsf{M}[r] = \mathsf{K}[r] + r \cdot \Delta$.
2. Output $(r, \mathsf{M}[r])$ to $\mathcal{P}$ and $\mathsf{K}[r]$ to $\mathcal{V}$.

Figure 1.1: Ideal functionality for generating VOLE.

communication in order to go through the circuit gate by gate and by the end of it the prover can open $[o]$ (where we let o be the output of the ciruit), to show that the input satisfies the circuit. Now, before we give a zero-knowledge protocol that uses the idea of commit and prove, we must first provide an additional functionality $\mathcal{F}_{\mathsf{PREP}}$ (Figure 1.2), which is used to instantiate $\mathcal{F}_{\mathsf{VOLE}}^{l,p}$ as well as allowing $\mathcal{P}$ and $\mathcal{V}$ to get so-called *multiplication triples* which are authenticated. These are sets $[a], [b], [c]$ so that $[c] = [a] \cdot [b]$. The triple-generation step follows the same pattern as the extension step from $\mathcal{F}_{\mathsf{VOLE}}^{l,p}$ (Figure 1.1), as it allows an adversarial prover or verifier to pick their own values, but apart from that ensures that the multiplication triples are correctly generated.

We now provide a simple zero-knowledge protocol Π_{zk} (Figure 1.3).

The keen-eyed reader may notice that this requires *getting* a triple $[a], [b], [c]$ so that $[c] = [a] \cdot [b]$. These can be generated in various ways, but usually comes at a cost, since it requires the prover and verifier to communicate, but at this point in time, techniques do exist to lower this cost tremendously. For now, we assume that there simply exists a way for $\mathcal{P}$ and $\mathcal{R}$ to get these, by using the functionality $\mathcal{F}_{\mathsf{PREP}}$ (Figure 1.2).

The above strategy does not utilize that $\mathcal{P}$ knows the underlying secrets. Recently, people started using this aspect to a higher degree, by simply allowing the prover to compute $z \leftarrow x \cdot y$ and then commit to z. The prover and verifier then push the verification of this to a final *verification* phase. An example of this is Wolverine [WYKW21a] in which the prover and verifier can avoid the expensive multiplication triples by using something they denote as *faulty triples*. These are multiplication triples that are generated by the prover and as such are potentially incorrect. It turns out, that if the verification phase use a *cut-and-choose* technique [BCC88] to check the multiplications computed during the circuit-evaluation, then the verifier will notice with very high probability, if

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

Initialize: On receiving Init from $\mathcal{P}$ and $\mathcal{V}$, send Init to $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$. $\mathcal{V}$ receives its global MAC key $\Delta \in \mathbb{F}_{p^\ell}$.

Extend: For (Extend), the parties send (Extend) to $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$. $\mathcal{P}$ receives $(\mathsf{M}[r], r) \in (\mathbb{F}_{p^\ell} \times \mathbb{F}_p)$ and $\mathcal{V}$ receives $\mathsf{K}[r] \in \mathbb{F}_{p^\ell}$ so that $\mathsf{M}[r] = \mathsf{K}[r] + \Delta \cdot r$.

Triple: On input (Triple) from $\mathcal{P}$ and (Triple) from $\mathcal{V}$,

1. If $\mathcal{V}$ is honest, then sample $\mathsf{K}[a], \mathsf{K}[b], \mathsf{K}[c] \leftarrow \mathbb{F}_{p^\ell}$. Otherwise receive $\mathsf{K}[a], \mathsf{K}[b], \mathsf{K}[c] \leftarrow \mathbb{F}_{p^\ell}$ from the adversary-
2. If $\mathcal{P}$ is honest, sample $a, b \leftarrow \mathbb{F}_p$, otherwise receive a, b the adversary. Let $c = a \cdot b$.
3. If $\mathcal{P}$ is honest, then compute $\mathsf{M}[a] = \mathsf{K}[a] + \Delta \cdot a$, $\mathsf{M}[b] = \mathsf{K}[b] + \Delta \cdot b$ and $\mathsf{M}[c] = \mathsf{K}[c] + \Delta \cdot c$. Otherwise receive $\mathsf{M}[a], \mathsf{M}[b], \mathsf{M}[c] \in \mathbb{F}_{p^\ell}$ from the adversary and recompute $\mathsf{K}[a] = \mathsf{M}[a] - \Delta \cdot a \in \mathbb{F}_{p^l}$, $\mathsf{K}[b] = \mathsf{M}[b] - \Delta \cdot b \in \mathbb{F}_{p^l}$ and $\mathsf{K}[c] = \mathsf{M}[c] - \Delta \cdot c \in \mathbb{F}_{p^l}$.
4. Output $((a, \mathsf{M}[a]), (b, \mathsf{M}[b]), (c, \mathsf{M}[c]))$ to $\mathcal{P}$ and $(\mathsf{K}[a], \mathsf{K}[b], \mathsf{K}[c])$ to $\mathcal{V}$.

Figure 1.2: Ideal functionality for preprocessing for the zero-knowledge protocol.

Protocol Π_{zk}

Preprocessing:

1. The prover and verifier send Init to $\mathcal{F}_{\mathsf{PREP}}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.
2. For $i \in \mathsf{in}$, $\mathcal{P}$ and $\mathcal{V}$ send (Extend) to $\mathcal{F}_{\mathsf{PREP}}$, returning authenticated values $[\lambda_i]$: $\{[\lambda_i]\}_{i \in \mathsf{in}}$.
3. For $i \in \mathsf{mul}$, $\mathcal{P}$ and $\mathcal{V}$ send (Triple) to $\mathcal{F}_{\mathsf{PREP}}$, obtaining $\{[a_i], [b_i], [c_i]\}_{i \in \mathsf{mul}}$ so that $c_i = a_i \cdot b_i$ for $i \in \mathsf{mul}$.

Online:

1. For $i \in \mathsf{in}$, $\mathcal{P}$ sends $\Lambda_i = w_i - \lambda_i \in \mathbb{F}_p$ to $\mathcal{V}$. Then both parties compute $[w_i] = [\lambda_i] + \Lambda_i$.
2. for each gate, $\mathcal{P}$ and $\mathcal{V}$ does the following:

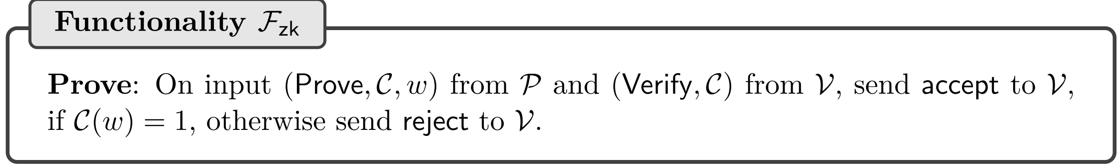
   Add On input $[x], [y]$ use the homomorphic addition property and output $[x] + [y]$.

   Mul On input $[x], [y]$ for the i'th multiplication gate, the prover and verifier do the following:

   (a) $\mathcal{P}$ and $\mathcal{V}$ computes and opens $e = [x] + [a_i]$ and $d = [y] + [b_i]$

   (b) Finally compute $[z] = [c_i] + e \cdot [y] - d \cdot [a_i]$ so that $z = ab + xy + ay - ya - ab = xy$
3. Finally, $\mathcal{P}$ and $\mathcal{V}$ run CheckZero$([w_o] - 1)$ where w_o is the value of the output gate.

Figure 1.3: The protocol for ZK proofs using the Commit and Prove technique


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

Prove: On input $(\mathsf{Prove}, \mathcal{C}, w)$ from $\mathcal{P}$ and $(\mathsf{Verify}, \mathcal{C})$ from $\mathcal{V}$, send accept to $\mathcal{V}$, if $\mathcal{C}(w) = 1$, otherwise send reject to $\mathcal{V}$.

Figure 1.4: Ideal functionality for ZK proofs.

the prover lied in either the multiplication *or* the faulty triples. More recently, one can use QuickSilver [YSWW21a] to generate correct multiplication triples, or simply apply QuickSilver directly to the multiplications from the prover. The technique described in Section 1.4.1 where we converted commitments based on VOLE into polynomials that could then be locally multiplied together, is very reminiscent of QuickSilver.

Now, as long as the commitments are *binding* and *hiding*, then it is clear that this way of evaluating a circuit guarantess *correctness*, *soundness* and *zero-knowledge*. Correctness follows from each gate being evaluated and the prover never lying, so each addition and multiplication will be correct. Soundness comes from the fact that the commitments are binding, so the prover can never change any of the underlying values and since the verification check (or the beaver triples) is/are correct, then the prover can never get through the check while having done anything dishonestly. Finally, zero-knowledge follows from the commitments being hiding. Informally, the verifier simply cannot tell which values are being worked on.

Now, to prove that the setup described above is secure and correct, we must first define what takes place in the ideal world. Here we have the functionality $\mathcal{F}_{\mathsf{zk}}$ (Figure 1.4) in which we consider a black-box which takes the input of both users and outputs either accept or reject, dependant on if $\mathcal{C}(w) = 1$ or not, respectively.

The functionality $\mathcal{F}_{\mathsf{zk}}$ (Figure 1.4) shows that both the prover and verifier agree on the circuit $\mathcal{C}$ that is being used, but only the prover knows the witness w.

Our goal is then to prove that we can describe a simulator Sim which interacts with an adversary and the functionality, while generating a view which is indistinguishable from the real interaction between a prover and a verifier.

Theorem 1. *Protocol Π_{zk} UC-realizes $\mathcal{F}_{\mathsf{zk}}$ in the $\mathcal{F}_{\mathsf{PREP}}$-hybrid model. In particular, no environment $\mathcal{Z}$ can distinguish the ideal world execution from the real world execution, except with probability at most ϵ_{open}.*

Proof sketch. We will first consider the case of a malicious prover (thus proving *soundness*) and then afterwards consider the case of a malicious verifier (thus proving *zero-knowledge* as well as showing *proof of knowledge*, a term we will encounter again in Section 1.6 during our introduction of SNARKs). In both cases we will construct a *probabilistic-polynomial-time* (PPT) simulator Sim which is given access to $\mathcal{F}_{\mathsf{zk}}$ and which will run the PPT adversary $\mathcal{A}$ as a subroutine while emulating the functionality $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ for $\mathcal{A}$. The simulator Sim will pass all communication between the adversary $\mathcal{A}$ and the *environment $\mathcal{Z}$.*

Malicious Prover Sim interacts with the adversary $\mathcal{A}$:

1. Sim emulates $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ for $\mathcal{A}$ by choosing a uniform $\Delta \in \mathbb{F}_{p^\ell}$ and keeping track of all the values $\{\lambda_i\}_{i \in \mathsf{in}}$ (and their corresponding MAC tags), which are used to commit to the input as well as $\{(a_i, b_i, r_i)\}_{i \in \mathsf{mul}}$ (and their corresponding MAC tags) which are used to commit to the multiplication triples, all of which are sent to $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ by $\mathcal{A}$. Sim then emulates $\mathcal{F}_{\mathsf{TRIPLES}}$ by recording $\{c_i\}_{i \in \mathsf{mul}}$ as well the corresponding MAC tags which are sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{TRIPLES}}$.

2. When $\mathcal{A}$ sends $\{\Lambda_i\}_{i \in \mathsf{in}}$, Sim sets $w_i = \lambda_i + \Lambda_i$ for $i \in \mathsf{in}$ (i.e. Sim now knows the witness)

3. Sim then runs the rest of the protocol as an honest verifier using the value Δ as well as the keys defined in the step 1. If the honest verifier outputs reject, then Sim sends $(\mathsf{Prove}, \mathcal{C}, \perp)$ to $\mathcal{F}_{\mathsf{zk}}$ and aborts. If the honest verifier outputs accept, then Sim sends $(\mathsf{Prove}, \mathcal{C}, w)$ to $\mathcal{F}_{\mathsf{zk}}$, using w as defined above.

We assume that $\mathcal{A}$ does not guess Δ correctly, which holds true except with probability $\epsilon_{\mathsf{open}} = \frac{1}{|\mathbb{F}|}$. It is then clear that the view of $\mathcal{A}$ is perfectly simulated by Sim, as, whenever the simulator Sim (simulating the verifier) outputs reject, then the real verifier outputs reject as well, since the simulator sends $\perp$ to $\mathcal{F}_{\mathsf{zk}}$. Lastly we need to consider the probability of the malicious prover winning, i.e. having the simulator outputs accept, despite the fact that the witness w sent by Sim results in $\mathcal{C}(w) = 0$, i.e. the witness *isn't* correct.

If $\mathcal{C}(w) = 0$, then it must be the case that $w_o = 0$. This follows from the addition being allowed through the additively homomorphic commitments and the multiplication being guaranteed through the correct multiplication triples in addition to the homomorphic commitments. Thus, all that remains is the probability of the adversary $\mathcal{A}$ opening some incorrect value by the end of the protocol, which happens with probability at most ϵ_{open}. This completes the proof for the case of a malicious prover.

Malicious verifier If Sim recieves reject from $\mathcal{F}_{\mathsf{zk}}$, then it simply aborts. Otherwise Sim interacts with the adversary $\mathcal{A}$ in the following way:

1. Sim emulates $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ by recording the global key Δ and the keys for all of the authenticated values which are sent to the functionality by $\mathcal{A}$. Sim then samples uniform values for $\{\lambda_i\}_{i \in \mathsf{in}}$ and $\{(a_i, b_i, r_i)\}_{i \in \mathsf{mul}}$ and compute their MAC tags.

2. Sim then executes steps 2-3 of Π_{zk} by simulating the honest prover using the input $w = 0^{\mathsf{in}}$ (i.e. using 0 for each value of the witness).

3. In the final step, Sim computes $\mathsf{K}[w_o]$ (using the keys send to $\mathcal{F}^{l,p}_{\mathsf{VOLE}}$ by $\mathcal{A}$) and then sets $\mathsf{M}[w_o] = \mathsf{K}[w_o] + \Delta$. Lastly, it uses $\mathsf{M}[w_o]$ to run $\mathsf{CheckZero}([w_o] - 1)$ with $\mathcal{A}$.

The view of $\mathcal{A}$ which is simulated by Sim, is distributed identically to its view in the real protocol execution, since the commitments to the input are *hiding*. □

This completes the proof of *soundness* and the proof of *zero-knowledge* of the zero-knowledge protocol Π_{zk} (Figure 1.3). Arguing *correctness* of the protocol is as simple as stating that the addition gates follows from the additively homomorphic commitments and that the multiplication gates are correct by the argument in the protocol. Thus, if the prover inputs a correct witness so that $\mathcal{C}(w) = 1$, then CheckZero will be satisfied.

1.5.1 Limiting Interaction

Whenever we have discussed protocols in the previous pages, there has been interaction between the prover and verifier. Either by the prover simply sending values to the verifier or the verifier sending *challenges* to the prover. It turns out that most zero-knowledge proofs follow this exact formular. While this works for some applications, other applications may struggle with this pattern, as they are directly harmed by the communication required, either through poor internet or simply the need for both the prover and verifier to be online at the same time. Fortunately, thanks to the seminal work of Amos Fiat and Adi Shamir [FS87], there exists a way of turning most interactive protocols into *non-interactive* protocols. These are attractive not only theoretically, but due to the reasons above, also practically.

Before introducing the technique brought forth by Fiat and Shamir, we must first discuss which types of protocols that can be transformed from interactive to non-interactive. Specifically, the construction must be a *public-coin* protocol. These are a special case of the general interactive proofs that were introduced in Section 1.1, in which the random choices made the verifier (e.g. when sampling the challenge for the prover) are made public or simply based on only public information. This idea was first introduced by Babai [Bab85] in the form of a game called the *Arthur-Merlin* game in which the verifier is played by the mere human Arthur whom has no special powers, while the prover is played by the almighty wizard Merlin who has unbounded resources (both properties that are brought into the game between the two). The game works the following way: Merlin sends the first message a_1 to Arthur who who then uniformly at random samples a challenge c_1 from a set of public random bit-strings B_1 (essentially flipping public coins), Merlin then sends a second message a_2 to which Arthur responds with c_2 sampled similarly from B_2 and so on. The game ends with a final message from Merlin, at which point Arthur either accepts or rejects, based on the input and the exchange of messages. The point here being that Merlin is unaware of exactly which challenges Arthur send before these have been sent, despite the challenges coming from publically flipped coins.

Now, Fiat and Shamir offered a generic transformation from a *public-coin* interactive protocol to a non-interactive protocol. This was achieved by replacing an honest verifier by having the prover hash its first message, using some hash function H, as well as a statement to derive the corresponding challenge that would have been publically sampled by the verifier, thus removing the need for interaction between the prover and verifier. This concludes with us essentially having a protocol only consisting of a prover, but then how do we prove security of it? Can we simply rely on the fact that if the original construction is provably secure, so is this new transformed version? Notably, how can we stop the prover from cheating and creating false proofs?

1.5.2 Modelling Security of The Fiat-Shamir Heuristic and Basic Usage

Formally, assume a proveably secure interactive protocol, denoted as Π, and let $\mathsf{FS}(\cdot)$ denote the Fiat-Shamir transformation. Can we then assume that $\mathsf{FS}(\Pi, H)$ is secure, provided that H is some standard cryptographic hash function? To answer this question, we must first provide some additional background. Thus far, we have only (albeit never mentioned this explicitly) considered the *standard model*. In this model, we only consider an adversary who is limited by amount of time and computational power. To this end, several works exist [GT03, BDG$^+$13] stating that it is indeed not the case, that we can simply assume that $\mathsf{FS}(\Pi, H)$ is secure. We therefore now consider a new model, the *random oracle model*. The random oracle model (or *ROM*) is an idea that was originally introduced by Bellare and Rogaway [BR93]. In this model we assume the existence of a so-called *random oracle* $\mathcal{O}$ which we informally describe in the following way:

- Anyone has access to the oracle $\mathcal{O}$
- On any distinct input, $\mathcal{O}$ returns a uniformly random bit string of fixed length
- $\mathcal{O}$ keeps track of every single input and will output the same bit string for the same input, regardless of who queried it.

Obviously, we cannot have such an oracle exist in real life. Additionally, it turns out that, even if we argue that something is secure in this model when we have access to the ideal oracle, things might still not work out when we try to instantiate the random oracle using a hash function, as shown by [CGH98, GT03]. Specifically, [CGH98] show that there exist signature and encryption schemes that are secure in the random oracle model, but fail for every concrete instantiation of any hash function. This shows that it is impossible to generically replace the random oracle used in the proof, with a hash function. This negative result shows that one has to be exceptionally careful when picking a function H by considering the inner-workings of the protocol Π. Despite this observation, the random oracle model has gained a large following and it has turned out to be trusted by cryptographers in general.

Now, the protocol Π_{zk} that was described in Figure 1.3 cannot properly utilize Fiat-Shamir by the simple fact that hardly any interaction takes place to begin with, since most of what happens, is the prover sending values to the verifier. This was done to make the protocol less complicated to explain, however, we must now go back on this slightly. To this end, we introduce a way of opening the committed-to values in a batch. This not only lowers the communication required, which would otherwise be two field elements per opening, but it also forces interaction into the protocol. The following approach is due to [WYKW21a].

The prover wishes to opening values n values $x_1, \ldots, x_n \in \mathbb{F}_p$ (which all has corresponding tags and keys).

1. $\mathcal{P}$ sends $x_1, \ldots, x_n$ to $\mathcal{V}$.

2. $\mathcal{V}$ picks uniform $\chi_1, \ldots, \chi_n \in \mathbb{F}_p$ and sends these to $\mathcal{P}$.

3. $\mathcal{P}$ computes $\mathsf{M}[x] = \sum_{i=1}^{n} \chi_i \cdot \mathsf{M}[x_i]$ and sends this to $\mathcal{V}$.

4. $\mathcal{V}$ computes $x = \sum_{i=1}^{n} \chi_i \cdot x_i \in \mathbb{F}_p$ and $\mathsf{K}[x] = \sum_{i=1}^{n} \chi_i \cdot \mathsf{K}[x_i] \in \mathbb{F}_p$. The verifier $\mathcal{V}$ accepts the opened values if and only if $\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x$.

The soundness of this approach follows from the following lemma

Lemma 1. *Let $x_1, \ldots, x_n \in \mathbb{F}_p$ and $\mathsf{M}[x_1], \ldots, \mathsf{M}[x_n] \in \mathbb{F}_p$ be arbitrary values known to $\mathcal{P}$, and let Δ and $\{\mathsf{K}[x_i] = \mathsf{M}[x_i] - \Delta \cdot x_i\}_{i=1}^{n}$, for uniform $\Delta \in \mathbb{F}_p$ be given to $\mathcal{V}$. The probability that $\mathcal{P}$ can successfully open values $(x'_1, \ldots, x'_n) \neq (x_1, \ldots, x_n)$ to $\mathcal{V}$ is at most $2/p$*

Proof. The proof of this can be found in [WYKW21a]. □

Now, note that this protocol requires the sending of random values $\{\chi_i\}_{i=1}^{n}$ from $\mathcal{V}$ to $\mathcal{P}$. A common trick to decrease the communication at this point, would be to instead have $\mathcal{V}$ sample a random *seed* and then send this over to $\mathcal{P}$, who can then sample these n values themself using a pre-selected hash function h and the seed. However, even this requires communication from the verifier to the prover.

Instead, we can use the Fiat-Shamir heuristic and then compute the set of coefficients $\{\chi_i\}_{i=1}^{n}$ as the output of a hash function *hash*, which is now evaluated on the values $\{x_i\}$ which are sent by $\mathcal{P}$ in the first step. Now, this can be shown ([WYKW21a]) to have a soundness error of at most $(q_H+2)/p$. In this case, q_H represents the number of queries to the random oracle that $\mathcal{P}$ makes, thus it is an upperbound on the hashes that are computed by the adversary.

To this end, we have introduced an interactive protocol Π_{zk} in Figure 1.3. We have then enhanced this protocol by adding a batch opening which allows the prover to open more values in cheaper fashion than if they were to be opened one-by-one. Lastly, we have argued that we can take this batch-opening protocol and make it completely non-interactive using the Fiat-Shamir heuristic as defined above.

1.6 SNARKs

So far we have primarily been discussing zero-knowledge protocols for circuits (which is also the type of protocol we described in Figure 1.3). These proofs are mostly linear in the communication taking place between the prover and verifier and so is the time it takes for them to do computations during it to either generate the proof or verify it. This naturally comes from evaluating the circuit gate-by-gate. This amount of communication works very well for smaller circuits, but it quickly becomes a potential problem for large circuits (many millions of gates). This is where *Succint Non-interactive Arguments of Knowledge* or (SNARKs) [BCCT12] comes into the picture. These are a type of proof system with some very specific properties. The name itself alludes to a few of these that we have not yet encountered.

Succinct The size of the proof is very small compared to the size of the statement or the witness. Specifically, it should be *sublinear* in the size of the witness or the statement. It is also very easy to verify and should take very little time.

Argument Back in Section 1.1 we described *soundness* as the property that a cheating prover $\mathcal{P}^*$ shouldn't be able to convince a verifier of a false statement. We now alter the scope of this slightly by (probably reasonably) assuming that the prover have bounded computational resources. If soundness only holds true for a polynomial-time $\mathcal{P}^*$, then a protocol is said to be an *argument* rather than a *proof.* Thus, for SNARKs, we consider it fair if a prover with unlimited compute power can create a SNARK that verifies without being correct.

of Knowledge For a prover to be able to construct a valid proof, there exists a so-called *extractor* that is capable of extacting a valid witness (i.e. the *knowledge*) for the statement. Thus, if the prover does not know a witness, the prover cannot construct a proof, as otherwise the extractor would be able to extract the valid witness.

The final property, them being non-interactive, is something we have already touched upon before during our discussion of Fiat-Shamir when applied to regular circuit-based proofs. The idea is the same for SNARKs. Finally, if the SNARK also satisfies zero-knowledge, we write zk-SNARK.

From this, we can conclude that zk-SNARKs are a particular type of proof system in which the proofs are very small, the verification process is fast, the proofs are non-interactive and we assume the prover is polynomially bounded in terms of computation. The latter however does not mean that it is cheap for a prover to create a zk-SNARK. In fact, this turns out to be one of the downsides of using zk-SNARKs compared to vole-zk as described above, it can be *very* costly both in terms of time but also the memory required by the prover [Tha22]. A direct comparison in terms of prover- and verifier time as well as communication and prover memory between a vole-zk protocol and several different zk-SNARK proof systems can be found in [WYKW21a].

This does however not mean that both proof systems aren't needed. Particularly, vole-zk occupies a different part of the overall space than existing zk-SNARKs. It balances the burden of work of the prover with the verifier as well as the communication, rather than put the vast majority of the work on the prover, which is exactly what zk-SNARKs do. This results in zk-SNARKs only being viable for small-scale computations or requiring expensive hardware. This is not to say that no constructions exist in which the prover uses only sublinear memory [COS20], but such schemes are currently much slower than those using linear memory [Set20].

Lastly, something worth discussing is the assumptions used by zk-SNARKs. Many constructions [BCC+16, BBB+18, WTs+18, CHM+20] are based on discrete-log type assumptions, which are assumptions that do not hold against a quantum polynomial-time adversary [Sho97]. What we mean by this, is that whenever (if ever?) a general-purpose quantum computer is built, it would render any construction based on these assumptions insecure. This is not to say that all zk-SNARK constructions will fail, just

that a lot of work has and is being put into this side of the field. Some constructions [GMNO18, BBHR18, AHIV17] are plausibly post-quantum. Work based on VOLE-based commitments are believed to be secure in the advent of a quantum compute, as they are based on variants of the Leaning Parity with Noise (LPN) assumption, which is believed to be secure post-quantum.

1.7 Applications

We consider different types of applications for the two different types of zero-knowledge proofs we have seen thus far: VOLE-based zero-knowledge over circuits and SNARKs. An important thing to note, that was not brought up in Section 1.6 on SNARKs, is that SNARKs can be *recursive*. A recursive SNARK allows the prover to *collect* (or *compose*) several SNARKs into a single SNARK whose size is smaller than if the composed SNARKs had simply been concatenated. Taking this idea slightly further, this can not only be used to collect many proofs into one, but also to prove slightly more complex statements. Essentially, if a prover proves a statement that results in a large proof (albeit still a SNARK), then the prover can create another SNARK to prove knowledge of the original proof, but now in a more compact fashion.

This idea of recursive SNARKs has a lot of applications in the realm of *blockchains* (or other applications utilizing Merkle trees [Mer79] and the likes). Here, one can consider an entire block of transactions which requires a proof that each of them are correct and valid. Having a SNARK for each would take up too much space, despite the succinctness of the individual proofs. Instead, one can split up the transactions in several subproofs and use a recursive SNARK to combine them all into a single proof. One could also consider a block that hasn't been completely filled up with transactions yet. Rather than wait for the entire block before beginning the proof, the prover can now *ahead of time* start creating the separate subproofs. This is known as *streaming* the proof generation and it results in a very small delay of the complete proof at the end. SNARKs fit these purposes very well, as one wants to minimize the size of the proofs on the blockchain, as these must be uploaded and communicated among many people, thus large proofs lead to much more communication. If one instead considers applications of VOLE-based zero-knowledge over circuits, these differ vastly from those of SNARKs. Here we highlight two: (1) proving properties of a program [CHP+23] and (2) proofs of machine-learning tasks [WYX+21a, LXZ21]. In (1) a prover wants to prove to a verifier that the prover knows an exploit for a specific program. This can be done by encoding both the program and the exploit to a circuit format (i.e. the exploit is encoded into the witness and the input to the circuit and the program is encoded into some verification circuit). The purpose of this type of proof would be to show that an exploit exist, for instance as part of a bug bounty program, to get the bounty before giving over the exploit. In (2) we consider machine-learning. A machine-learning model can require huge effort to train and because of this, the owner may only provide the model as a paid service. Because of this, whoever wants to use this model must send their input to the owner who then runs the model and sends back the answer. To avoid the need of this trust-based system, the

owner of the model can, using a scalable zero-knowledge protocol, publicly commit to the machine-learning model and then prove in zero-knowledge that the committed model was correctly applied to the users input which yielded the result that was previously claimed.

Both of these use-cases potentially require millions (or billions) of gates to model either the program or the machine-learning model. This quickly becomes far too large for a SNARK to handle, as the time and memory consumption of the prover would grow beyond what is reasonable (or possible in some cases).

Instead, VOLE-based proofs are much more scalable, since the computational resources required by the prover and verifier are roughly the same as what would be needed to simply evaluate the statement using the witness in the clear (i.e. the prover gives the witness to the verifier). In addition to this, the memory required by the prover, when considering VOLE-based proofs, is roughly proportional to the memory required to simply verify the statement when given the witness. This turns out to be incredibly useful, as the witness can sometimes be so large that it does not fit in memory, despite the proof being able to be verified efficiently in a streaming manner. On the other hand, VOLE-based proofs would be quite bad for blockchain applications, as they require significantly more communication (or if we only consider non-interactive proofs, then they result in significantly larger proofs). In addition to this, VOLE-based proofs are not necessarily the best choice if the verifier has to be able to verify a proof quickly (like they can with SNARKs), as the verification time usually scales linearly with the size of the circuit.

In conclusion, both SNARKs and VOLE-based ZK proof systems have a reason to exist, but these reasons do not overlap, as they solve entirely different problems.

1.8 Overview

As seen in the introduction, we can build both interactive as well as non-interactive zero-knowledge protocols based on linearly-homomorphic commitments that we build from information-theoretic (IT) MACs. As also shown, these IT MACs can be built efficiently from vector oblivious linear-function evaluations when we are working over a large prime field $\mathbb{F}_p$. The goal of this thesis is to push forward the study of cryptographic constructions based on this particular type of commitment in multiple directions.

The main body of this thesis focus on multiple things

1. Optimising and expanding on what is possible using circuit-based cryptography by increasing the number of settings in which we can efficiently use circuit-based zero-knowledge proofs based on VOLE as well as providing gadgets that lower the overall gate-count.

2. Optimising protocols in the so-called *Random Access Memory* (or RAM) model.

1.8.1 Expanding on the possibilities of circuit-based cryptography

In this thesis we investigate the possibilities of going beyond the gate-by-gate paradigm of circuit-based cryptography. When one evaluates a circuit gate-by-gate it naturally brings with it a linear running time in the size of the circuit. While this is fine for smaller circuits, it can be detrimental when the circuits consist of millions or billions of gates, if the constants related to the linear costs are too high.

A recent line of work has been investigating the scalability of zero-knowledge protocols for *very* large statements which are represented by circuits with billions of gates [WYKW21a, BMRS21a, DIO20, YSWW21a, DILO22]. This is primarily done by building certain gadgets that can do specific often-used primitives more efficient, than if they were naïvely implemented using a regular circuit or by utilizing the structure of the computation coupled with techniques from polynomial arithmetics, allowing certain operations to be performed more efficiently than linearly in the number of gates. In the first case, the goal is to decrease the overall number of gates required, while the second case requires looking at the problems in a different way. A third approach to making zero-knowledge more efficient, is considering other domains of computation, as some functions are known to be *more efficient* when expressed as a circuit over a particular domain.

In this thesis we propose new gadgets for often used techniques, by giving protocols for integer truncation, integer comparison and field switching. While integer truncation and integer comparison are self-explanatory, field switching is a bit more obscure. It turns out, that there are operations that are much more efficient when represented in a specific field. For instance, comparisons or bit operations are more efficient when expressed over $\mathbb{F}_2$ while integer arithmetics best fits into $\mathbb{F}_p$ for some prime p. The problem with this, is that most state-of-the-art zero-knowledge compilers operate over a single domain. If this is $\mathbb{F}_p$, then any arithmetics over $\mathbb{F}_2$ will require costly bit-decompositions and thereafter

emulation of the binary circuit logic in $\mathbb{F}_p$. This is in particular where being able to switch between fields turns out to be very useful.

Meanwhile, neither of the above mentioned fields capture arithmetics modulo 2^k efficiently. When working over the *ring* $\mathbb{Z}_{2^k}$ most protocols that are designed to work over $\mathbb{F}_p$ turn out to be insecure due to properties of working over a field no longer being present. It is however of recent interest, to be able to work over the ring $\mathbb{Z}_{2^k}$. For instance, since CPUs perform arithmetic in $\mathbb{Z}_{2^k}$, working in this ring leads to a much simpler translation of program code into a satisfiable circuit $\mathcal{C}$ for which a prover can prove knowledge of a satisfying witness.

All of the chapters in this section has in common that they use VOLE, but apart from that they strive to improve performance of circuit-based zero-knowledge in various directions, either by proposing optimizations to gadgets that are often used in protocols or by allowing the protocols to work more efficiently over various domains such as $\mathbb{F}_2, \mathbb{F}_p$ or $\mathbb{Z}_{2^k}$

Chapter 2: *Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs over* $\mathbb{Z}_{2^k}$

This chapter is based on the following published work [BBMHS22a]. The results were also presented at CRYPTO 2022. For the full version, including appendices see [BBMHS22b] (this thesis contains a version of this where the appendices have been moved to their locations within the paper).

Background This chapter considers ways of generating vector-OLE (VOLE) over rings $\mathbb{Z}_{2^k}$ as well as zero-knowledge protocols working over this domain. Lately, a lot of effort has been put in to produce VOLEs over finite fields $\mathbb{F}_p$ for some large prime p [BCGI18, BCG+19b, WYKW21a]. All of these works rely on the Leaning-Parity with Noise (or LPN) assumption, albeit efficient, their protocols would be insecure when run over a ring $\mathbb{Z}_{2^k}$. As such, the most efficient way of generating this correlated randomness over $\mathbb{Z}_{2^k}$ (while maintaining an actively secure protocol), would instead be [Sch18]. This paper builds on the idea of *OT-extension* in which the prover and verifier generate a small set of correlated OTs in an expensive setup phase and then *extend* these into many more, through a much cheaper extension phase. The paper however, only uses techniques that work both over rings as well as fields, while still being actively secure, thus allowing for the generation of VOLE over the ring $\mathbb{Z}_{2^k}$. Unfortunately, this has the drawback of requiring quadratic communication in the bit length of the VOLE.

On a somewhat related note, the above discussion focuses on protocols having *active* security. If one is only interested in *semi-honest* security (and multi-party computation), other works do exist [SGRR19, BCG+19b] for generating correlated OTs or VOLEs over rings $\mathbb{Z}_{2^k}$.

There has been little to no work on protocols for circuits over $\mathbb{Z}_{2^k}$ in the commit-and-prove setting however. Prior work by Baum et al.[BBMH+21b] (Chapter 3 of this thesis), have shown how to adapt VOLE-based information-theoretic MACs for zero-knowledge over $\mathbb{Z}_{2^k}$, using techniques from multiparty-computation [CDE+18]. They then present

two variations of zero-knowledge protocols that uses the new information-theoretic MACs as well as an (adapted to $\mathbb{Z}_{2^k}$) multiplication check from Wolverine [WYKW21a] in the first variation and one which uses a multiplication check (also adapted to $\mathbb{Z}_{2^k}$) from Mac'n'Cheese [BMRS21a]. More recently, a more efficient zero-knowledge protocol called QuickSilver [YSWW21a] was presented, but this unfortunately only works over finite fields.

Contribution Firstly, this work addresses the question of building a linearly-homomorphic commitment scheme over $\mathbb{Z}_{2^k}$ and in turn builds efficient protocols for generating VOLE over the ring $\mathbb{Z}_{2^k}$. It succesfully builds a maliciously secure protocol with efficiency that is comparable to the state-of-art over finite fields [BCG+19a, WYKW21a]. The protocol for generating VOLE is rooted in [WYKW21a], but it required the introduction of a new consistency check for verifying correctness of the VOLE extension, which is tailored specifically to handle any challenges which comes with working over $\mathbb{Z}_{2^k}$, such as dealing with additional leakage during the protocol for single-point VOLE. This requires a new assumption that we denote as *Leaky Regular LPN*. This is a variant of the *Regular LPN* assumption which leaks part of the error vector, but we show that this doesn't hurt the overall hardness of the problem, as long as certain measures are taken.

Secondly, this work shows how to adapt QuickSilver [YSWW21a] to work over $\mathbb{Z}_{2^k}$. This uses the aforementioned VOLE protocol and is able to prove circuit satisfiability over $\mathbb{Z}_{2^k}$, by lifting ideas from the world of multi-party computation [CDE+18] as well as careful analysis of the roots of the used polynomials. In doing so, we obtain an efficient zero-knowledge protocol that we call QuarkSilver, yielding comparable efficiency to QuickSilver.

Chapter 3: *Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and* $\mathbb{Z}_{2^k}$

This chapter is based on the following published work [BBMH+21b]. The results were also presented at CCS 2021. For the full version, including appendices see [BBMH+21a] (this thesis contains a version of this where the appendices have been moved to their locations within the paper).

Background As some operations are much more efficient when working with bits rather than large field elements, a much used operation is computing bit decompositions of field elements. The tried and tested way of converting committed-to values between their binary decomposition and the value itself, in some prime field $\mathbb{F}_p$ is to have $\mathcal{P}$ compute, as well as commit to, the binary decomposition of the hidden value $x \in \mathbb{F}_p$, denoted by $[x_0]_p, \ldots, [x_{m-1}]_p$ and then prove that not only are each of the values truly bits, but also that $x = \sum_{i=0}^{m-1} 2^i x_i$. When using a linearly homomorphic commitment scheme, such as the one achieved by using VOLE, the latter operation comes for free. The first proof however, proving that each field element is a bit, turns out to be very costly. The prover must show that $x_i \cdot (x_i - 1) = 0 \; \forall i$, and multiplications require com-

munication, when using the aforementioned linearly homomorphic commitment scheme. Now, equipped with the proven bit decomposition of some value $x \in \mathbb{F}_p$, one still has to mimic the logic of whatever binary circuit had to be computed to begin with, requiring further work and communication. Additionally, before this computation can take place, this requires communicating the bit decomposition in $\mathbb{F}_p$. Consider a 60-bit integer committed to in some prime field $\mathbb{F}_p$, doing the above will require the prover to commit to an additional 60 values in $\mathbb{F}_p$, in addition to what it requires to check consistency of the multiplications.

As briefly touched upon in Section 1.2, it is sometimes desired to work over binary representations i.e. $\mathbb{F}_2$ or when having the proven bit decomposition in $\mathbb{F}_p$, despite the heavy costs listed above. This is because it can allow for much more efficient circuits than when working over the value $x \in \mathbb{F}_p$ rather than the bit decomposition $[x_1]_p, \dots, [x_m]_p$. A particular example is truncation. Consider some committed-to value $[x]_p$, $x \in \mathbb{F}_p$ for some prime p as well as its bit decomposition $([x_0]_p, \dots, [x_{m-1}]_p)$ (where $\mathcal{P}$ knows the underlying values and $\mathcal{V}$ *does not*). Now, computing a truncation of the hidden value $[x]_p$ requires both proving the size of the underlying value (i.e. its bit-length) and that the correct number of bits were truncated off. If instead the prover have previously committed to the bit decomposition $([x_0]_p, \dots, [x_{m-1}]_p)$, then computing the truncation of l bits is as simple as outputting $([x_0]_p, \dots, [x_{m-1-l}]_p)$. This can be done locally without communication. This operations becomes even more efficient if working over $\mathbb{F}_2$ is an option, as communicating the values $([x_0]_2, \dots, [x_{m-1}]_2)$ would require less communication than the same values over $\mathbb{F}_p$.

Contribution In this chapter we present a new way of converting between values from $\mathbb{F}_p$ (or $\mathbb{Z}_{2^k}$) and the boolean field $\mathbb{F}_2$ in zero-knowledge, that is inspired by research from secure multiparty computation [EGK+20]. We find several optimizations as well as the need for a consistency check during the adaption of the protocol to zero-knowledge. This new conversion check results in the prover and verifier having values $[x]_M$ and $([x_0]_2, \dots, [x_{m-1}]_2)$ so that $x = \sum_{i=0}^{m} 2^i \cdot x_i$. Therefore, since this conversion protocol can be run directly as a sub-protocol within a circuit, it removes the need for ever emulating the logic of binary circuits in $\mathbb{F}_p$ or vice versa, as one can compute directly within $\mathbb{F}_2$ and then later on convert the binary values back into $\mathbb{F}_p$. Additionally, the conversion protocol removes the need for ever having the bit decomposition committed to in $\mathbb{F}_p$. We then show how to use this new conversion protocol, as a way to compute truncation of field elements in $\mathbb{F}_p$ as well as compute comparisons. The idea is that when we show that a value $[x]_M$ can be represented by m bits $([x_0]_2, \dots, [x_{m-1}]_2)$, then we indirectly show that $x < 2^m$ as well, i.e. it also works as a range proof.

Finally, while protocols over both $\mathbb{F}_p$ and $\mathbb{F}_2$ are well-studied, there has been less work on protocols over the ring $\mathbb{Z}_{2^k}$. This paper takes a step in this direction by designing the above protocols such that they also work over a ring $\mathbb{Z}_{2^k}$. In addition to this, it is also the first paper to show how to use the VOLE-based commitments for ZK over $\mathbb{Z}_{2^k}$. This paper present two variations of zero-knowledge protocols that uses the new information-theoretic MACs as well as an (adapted to $\mathbb{Z}_{2^k}$) multiplication check from

Wolverine [WYKW21a] in the first variation and one which uses a multiplication check (also adapted to $\mathbb{Z}_{2^k}$) from Mac'n'Cheese [BMRS21a]. This provides flexibility in the high-level application of these protocols. Here it is worth mentioning that this result was later improved on in Mozzarella [BBMHS22a] (Chapter 2 of this thesis).

Chapter 4: *Cheddar – Oh, Range Proofs for VOLE-based Zero-Knowledge!*

This paper is currently a draft and has as such not been published anywhere yet.

Background As we have already covered, zero-knowledge proofs allow a prover to prove to a verifier that a specific statement is true while revealing no additional information. This idea works for arbitrary functions that the prover wants to prove and these functions can be modelled as general circuits. However, rather than use general circuits (and naïve implementations), oftentimes protocols designed for specific purposes are significantly more efficient. One of these types of protocols are *Range Proofs*. A range proof allows a prover to prove to a verifier that some committed-to value x lies in some public range $a \leq b$ so that $a \leq x \leq b$.

Range proofs are a fascinating primitive, as they can be used alone in applications such as [Cha90], e-voting [Gro05, ABG+21], e-cash [CHL05], or lattice-based cryptography in general where security and correctness is based on the smallness of certain integers and vectors [CMM19, LLM+16, GHL22]. Also of interest in recent years is verifiability of large-scale neural network inference [LXZ21]. The main difficulty here lies in the fact that a lot of non-linear operations cause a loss in accuracy when the values are represented as fixed-point numbers, something that the linear operations require, so range proofs can be used to convert between fixed-point numbers and floating-point numbers, which can then be utilized in the protocols to obtain maximum accuracy.

A recent technique due to Baum et al.[BBMH+21b] (Chapter 3 of this thesis) build a range proof by using their field switching protocol. A consequence of proving correctness of bit decompositions is that it shows what the bit length of a value x is at most. At the time, this was the state-of-the-art technique, however, it suffers from requiring *a lot* of range proofs at a time, to be efficient. Specifically, it is at its best efficiency when checking 2^{20} range proofs at a time. This comes from requiring a cut-n-choose step, that is insecure below this threshold. In addition to requiring many at a time, it also only allows a fixed range for the entire batch.

Protocols for range proofs usually fall into two main paradigms: (1): range proofs based on n-ary decomposition [Gro11, CCs08, BBB+18, BBMH+21b], or (2): range proofs based on square decomposition [Bou00, Lip03, Gro05, CPP17, GHL22]. Square decomposition involves the prover and verifier computing $[x] - a$ and $b - [x]$ and then for the prover to decompose the product of $x - a$ and $b - x$ into four squares so that $([x] - a)(b - [x]) = \alpha^2 + \beta^2 + \gamma^2 + \delta^2$ [Lip03] and thus proving that both numbers are positive, which shows that $a \leq x \leq b$. This is usually done using an integer commitment scheme so that no overflow can happen during the computations but it can also involve an approximate range proof where the prover shows that the values $(\alpha, \beta, \gamma, \delta)$ are all

bounded so by some predetermined bound β. While the approximate range proof is much more efficient, it also has a slack that allows the values to be off by at most this slack, while the proof still succeeds. This slack is based on the number of values involved in the proof and usually means the values must either be small or few values can be used in each proof [BN20, GHL22]. The former, n-ary decomposition involves the prover decomposing x into field elements (such as bits in the case of *binary* decomposition) and then prove that each of those fit within a predetermined range as well as the decomposition being a correct decomposition. The aforementioned work due to Baum et al.[BBMH+21b] also fall into this category.

Now, the different approaches each have their pros and cons. Particularly, computing the square decompositions becomes a bottle neck when considering two large values or two many values. They therefore works best on smaller values and smaller batches of range proofs. They do however not require a lot of communication, as the primary communication is committing to the squares computed by the prover. The other approach of n-ary decomposition has the problem of potentially requiring more communication, as the decomposition has to be communicated (and committed to). It is however relatively cheap to compute. Depending on the value of n, this approach works better for slightly larger batches, compared to square decomposition, as some techniques can be used to more efficiently perform the equality checks required by this approach, when the batches aren't too small. Finally, we put the approach of [BBMH+21b] in its own category, as this approach communicates the decomposition in $\mathbb{F}_2$ rather than $\mathbb{F}_p$, thus saving on communication here. This saving however, requires a very large batch of range proofs at a time or else it becomes efficient significantly more inefficient.

Contribution This chapter provides three novel ways of generating range proofs where each way has its pros and cons. The first way utilizes the square decomposition, but we avoid the expensive part of requiring an integer commitment scheme by instead using a significantly cheaper approximate range proof on each value $(\alpha, \beta, \gamma, \delta)$ and thus show that the values won't overflow when squared or added together. Let $\mathbf{s} \in \mathbb{F}_q$ be a secret vector (that the prover has committed to) and $\mathbf{C}$ be a public matrix sampled by the verifier. The way the approximate range proof is usually done is by computing $[\mathbf{y}] = \mathbf{C}[\mathbf{s}]$ and then show that $[\mathbf{y}]$ is still small. We show that by sampling the matrix $\mathbf{C}$ according to particular distribution, we can lower the slack caused by the approximate range proof drastically, thus allowing us to run our protocol on either many more values or larger values than related work.

The second range proof we propose is in the n-ary decomposition paradigm. The prover has committed to x and claims that $0 \leq x \leq 2^k - 1$ for some public value k. Additionally, the prover and verifier have agreed on some value n for which the prover computes $x_0, \ldots, x_{(k/n)-1}$ so that $x = \prod_{i=0}^{k/n} 2^{i \cdot n} x_i$. The prover then defines the polynomial $f(X) = \prod_{i=0}^{2^n-1} i - X$ and checks correctness of the polynomials. The primary cost of this proof comes from checking the correctness of the polynomials, which can be done using a variation of QuickSilver [YSWW21a]. Instead of having to write polynomials in a *degree-separated* format as well as using polynomial operations, we

propose seeing the commitments as 1-degree polynomials and then combine d of these 1-by-1 to generate a degree d polynomial. Note now that the degree d term will consist only of the underlying values $x_0, \dots, x_{d-1}$, which must evaluate to 0, if the prover is honest. Thus, we can mask the remaining polynomial of degree $d-1$ and send this over so that the verifier can check this is done correctly. This simplifies the protocol and for the simply polynomials we use, we expect this to be faster. A second approach is to use Mac'n'Cheese [BMRS21a], who proposed a disjunction proof, which can be used to show that one of the values $x_0, \dots, x_{d-1}$ is 0 and thus the values fit.

The final proof is also in the n-ary decompositional paradigm. Given commitments $[a_1], \dots, [a_n]$ over a large finite field $\mathbb{F}_q$, the prover wants to show that all $a_i \in [0, 2^m)$ for some $m < \log_2(|\mathbb{F}_q|)$. That is, they can be represented as $a_i = \sum_{j=1}^{m} 2^{j-1} \cdot b_{i,j}$ with all $b_{i,j} \in \{0, 1\}$. The prover then computes polynomials $p_1, \dots, p_m, q_1, \dots, q_m \in \mathbb{F}_q[X]$ such that $p_j(i) = b_{i,j}$ and $q_j(i) = 1 - b_{i,j} = 1 - p_j(i)$ for $i \in [n], j \in [m]$ but through certain techniques can avoid ever sending over the entire decompositions.

1.8.2 Considering the RAM model

Statements presented as circuits, either Boolean or arithmetic is what this thesis is primarily about. Unfortunately, this approach can fall short in instances such as set membership, private database search, verification of program execution or something as simple as binary search. In these particular cases, we can utilize a different type of computation. Specifically, random-access-machines (RAM) rather than circuits. Consider the case of binary search. Here a RAM that searches for a specific value in a sorted list of n values can do so in only $\log n$ memory accesses. This is in contrast to the canonical approach using circuits where it is at least required for the circuit to parse the entire list as input after which comparisons can be made to find the correct value. This means that the size of the circuit at least depends linearly on the size of the list. When we want to utilize this notion of RAM in zero-knowledge proofs, we consider the RAM model.

In this model, we consider the case of the prover and verifier having access to a shared memory of size N. The prover can store values at certain addresses in the memory and can then afterwards read the values from those same addresses. Key to this model though, is the verifier can never know which operation is being performed and can thus not know which addresses are getting read or written to, or which value is being read or written. Meanwhile, the verifier can keep track of the fact that nothing is being read that has not been written.

In general, the RAM model is preferable to using circuits whenever the computations involve sparse data-dependent accesses across very large datasets, as the RAM model allows reading from and writing to independent addresses across a large shared array. In addition to this, it also has the benefit of most real-world computations already being expressed in terms of RAM machines, thus allowing zero-knowledge proofs in the RAM model to handle statements that are written in ordinary programming languages [HYDK21]. This opens the door for allowing a broad audience to use widely available tools to build efficient proofs of arbitrary statements written in regular programming languages, rather than having to use obscure zero-knowledge or multiparty computa-

tion frameworks. As such, it is necessary for zero-knowledge proofs in the RAM model to be as efficient as possible, so recently the RAM model has seen a lot of interest [BCG+13, BFR+13, BCTV13, HMR15, MRS17, BCG+18, HK20a, FKL+21, DOTV22]

Chapter 5: *Pecorino: More Efficient Zero-Knowledge for RAM Programs*

This paper has not been published yet and is currently not on eprint.

Background Developing protocols for zero-knowledge proofs in the RAM model has been of interest for a few years at this point. Early on, protocols would use a permutation network in order to enfore consistency of memory accesses [BCG+13, BCTV13, WSH+14] within the shared memory. This permutation network would take two lists, denoted by tr and $\mathsf{tr}_{\mathsf{sort}}$, where tr essentially is a list of tuples containing operations, values, addresses as well as timestamps and $\mathsf{tr}_{\mathsf{sort}}$ is tr sorted on addresses followed by timestamps. The task of the permutation network is then to check that these two lists contain the same values and operations by showing that $\mathsf{tr} = \pi(\mathsf{tr}_{\mathsf{sort}}$ for a permutation π. This comes at a cost though, as the network used [Ben64] has a logarithmic overhead per memory access. Recently, [FKL+21], managed to get rid of this expensive permutation network, by instead utilising a polynomial-based permutation check. They still have the two lists tr and $\mathsf{tr}_{\mathsf{sort}}$ containing the trace of memory accesses and the same list which is (supposedly) sorted, respectively. They then pack each tuple into a single value, by utilizing the structure of their commitments (specifically they use VOLE based commitments), and as a result get two lists of field elements. They then define two polynomials

$$p(X) = \prod_i (\mathsf{tr}[i] - X) \qquad q(X) = \prod_i (\mathsf{tr}_{\mathsf{sort}}[i] - X)$$

and simply check equality between the two polynomials by evaluating them on a random value r, i.e.

$$p(r) \stackrel{?}{=} q(r),$$

which, when using techniques due to QuickSilver [YSWW21a], is much faster and requires little communication, compared to using the permutation network. These protocols do not translate well to non-interactive protocols, as they are private-coin (i.e. cannot be transformed using the Fiat-Shamir heuristic). More recently, de Saint Guilhem et al.[DOTV22] proposed a public-coin constant-overhead zero-knowledge protocol in the RAM model which can be transformed using the Fiat-Shamir heuristic, as they transform any public-coin zero-knowledge system into a proof of RAM programs. This work [DOTV22] is based on the work due to [FKL+21], but in addition to being potentially non-interactive, their protocol is also the first protocol allowing values from a prime field, rather than only supporting boolean values.

Taking a step back; the purpose of this sorting network used by [FKL+21], is to sort two lists containing tuples (value, address, timestamp, operation). By sorting these entries on address and then timestamp, the authors can ensure that the prover $\mathcal{P}$ performs

all reads after a write has been performed on a particular address. While the sorting is not the expensive part, ensuring that it's sorted correctly and that the operations match in regards to having reads after writes, is quite expensive. Specifically, the check involves checking that for all adjacent tuples ($[l_i], [d_i]$ and ($[l_{i+1}], [d_{i+1}]$) (where each tuple contains an address and value, respectively), either $l_i < l_{i+1}$ or $l_i = l_{i+1}$ and $d_i = d_{i+1}$. This represents the case when the shared memory is only readable and not writeable. In the latter case, the check requires extra if-statements to check that the operations match.

Contribution In this chapter we present our new ZK-RAM protocol with linear overhead in both the word size and the circuit complexity. The key idea of our approach is to avoid using the expensive sorting network which is used to sort the entire memory as part of the consistency check. The purpose of the check is to ensure that read and write operations on each address are performed in correct order (i.e. the prover must write to an address before reading it and these must be done in correct order). After the sorting has been done by the prover, the prover must additionally prove that the sorting is done correctly. Both of these checks require several multiplications as part of several comparisons, as each adjacent entry must be checked. This sorting network or some variant thereof is used by all previous work. Instead of using this sorting network, we *assert* that a set of read operations equals a set of write operations by keeping track of each performed operation in new manner. In order to keep track of the order of operations, instead of timestamps (which are used as tiebreakers after having sorted on addresses), we assert the order of operations by using challenges supplied by the verifier. For each write, the verifier sends over a single field element which is used as this challenge. Now, if the prover wants to read this address again, the prover must commit to this challenge. As these are all unique with probability $1/|\mathbb{F}|$, the prover can't guess these in advance, and as such cannot ever read an address that has not been written to first, as the prover would have to guess the challenge. Additionally, to make this trick work, we introduce a new primitive called ReadAndWriteBack. During this operation, the verifier sends the random challenge that the prover writes into an array keeping track of all write operations. Whenever a read operation takes place, the prover then writes the same randomness value (in addition to other information such as the value and address of the read) into an array keeping track of the read operations. The verifier also does this, except on committed-to values. This results in a protocol which is roughly 5× more efficient than [FKL+21] in terms of time spent per access.

Finally, the prover and verifier assert equality between the two arrays. Now, the prover can never have read a value before it was written, as that would require the prover to guess which value the verifier would send over during the write operation. This allows us to skip the sorting network in its entirety. Finally we also show how to make our protocol non-interactive by applying the Fiat-Shamir heuristic to compute the random values sent over by the verifier during the interactive variant.

1.9 Additional Publications

In addition to the works described in this thesis, the author has contributed to the following result [MHOY21] which proposes the first threshold ring signature where the size of the signatures scale linearly with the number of signers t.

1.9.1 Stronger Notions and a More Efficient Construction of Threshold Ring Signatures

[MHOY21] Alexander Munch-Hansen, Claudio Orlandi, and Sophia Yakoubov. Stronger no- tions and a more eicient construction of threshold ring signatures. In Patrick Longa and Carla Ràfols, editors, LATINCRYPT 2021, volume 12912 of LNCS, pages 363381. Springer, Heidelberg, October 2021. Full version available at https://eprint.iacr.org/2020/678.pdf

Part II

Publications

Chapter 2

Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs Over $\mathbb{Z}_{2^k}$

2.1 Introduction

Zero-knowledge (ZK) proofs allow a prover to convince a verifier that some statement is true, without revealing any additional information. They are a fundamental tool in cryptography with a wide range of applications. A common way of expressing statements used in ZK is with *circuit satisfiability*, where the prover and verifier hold some circuit $\mathcal{C}$, and the prover proves that she knows a witness w such that $\mathcal{C}(w) = 1$. Typically, $\mathcal{C}$ is an arithmetic circuit defined over a finite field such as $\mathbb{F}_2$ or $\mathbb{F}_p$ for a large prime p, but the same idea works for any finite ring.

A recent line of work [WYKW21a, YSWW21a, BMRS21a, DIO21a] builds highly scalable zero-knowledge proofs based on vector oblivious linear evalution, or VOLE. VOLE is a two-party protocol often used in secure computation settings, which allows a receiver holding Δ to learn a secret linear function $\mathbf{w} - \Delta \cdot \mathbf{u} = \mathbf{v}$ of a sender's private inputs $\mathbf{u}, \mathbf{w}$. VOLE-based ZK protocols have the key feature that the overhead of the prover is very small: compared with the cost of evaluating the circuit $\mathcal{C}$ in the clear, few additional computational or memory resources are needed. This allows proofs to scale to handle very large statements, such as proving properties of complex programs. On the other hand, potential drawbacks of using VOLE are that the communication complexity is typically linear in the size of $\mathcal{C}$ – unlike SNARKs (e.g. [MBKM19, BBHR19]) and MPC-in-the-head techniques (e.g. [AHIV17]), which can be sublinear – and proofs are only verifiable by a single, designated verifier.

VOLE Constructions. In a length-n VOLE protocol over some ring R, the sender has input two vectors $\mathbf{u}, \mathbf{w} \in R^n$, while the receiver has input $\Delta \in R$, and receives as output $\mathbf{v} \in R^n$ as defined above. In applications such as ZK proofs, it is actually enough to construct *random VOLEs*, or VOLE correlations, where both parties' inputs are chosen at random. The most efficient approaches for generating random VOLE are based on the method of Boyle et al. [BCGI18], which relies on an arithmetic variant of the learning parity with noise (LPN) assumption. The protocol has the key feature that the communication cost is *sublinear* in the output length, n.

The original protocol of [BCGI18] has only semi-honest security (or malicious security using expensive, generic 2-PC techniques). Later, dedicated maliciously secure protocols over fields were developed [BCG+19a, WYKW21a], which essentially match the cost

of the underlying semi-honest protocols, by using lightweight consistency checks for verifying honest behavior. In general, these protocols assume that R is a finite field.

ZK Based on VOLE. The state-of-the-art, VOLE-based protocol for proving circuit satisfiability in zero-knowledge is the QuickSilver protocol. QuickSilver, which builds upon the previous Line-Point ZK [DIO21a] protocol, works for circuits over any finite field $\mathbb{F}_q$, and has a communication cost of essentially 1 field element per multiplication gate. Concretely, QuickSilver achieves a throughput of up to 15.8 million AND gates per second for a Boolean circuit, or 8.9 million multiplication gates for an arithmetic circuit over the 61-bit Mersenne prime field. Another approach is the Mac'n'Cheese protocol [BMRS21a], which can also achieve an amortized cost as small as 1 field element, but with slightly worse computational costs and round complexity.

ZK Over Rings. While most ZK protocols are based on circuits over fields, it can in certain applications be more desirable to work with circuits over a finite ring such as $\mathbb{Z}_{2^k}$. For instance, to prove a property of an existing program (such as proving a program contains a bug, or does not violate some safety property) the program logic and computations must all be emulated using a circuit. Since CPUs perform arithmetic in $\mathbb{Z}_{2^k}$, this is a natural choice of ring that leads to a simpler translation of program code into a satisfiable circuit $\mathcal{C}$.

Unfortunately, not many existing ZK proof systems can natively support computations over rings. The recent work of [BBMH+21b] gave the first ZK protocol over $\mathbb{Z}_{2^k}$ based on VOLE over $\mathbb{Z}_{2^k}$, obtaining a proof system with a communication cost of $O(1)$ ring elements per multiplication gate (for large rings), asymptotically matching QuickSilver over large fields. However, a major drawback of their protocols is that they require maliciously secure VOLE over $\mathbb{Z}_{2^k}$, which is much more expensive to build: the only known instantiation of this [Sch18] would increase the concrete communication of their ZK protocol by 1–2 orders of magnitude. Finally, another approach to zero-knowledge proof systems over rings has been proposed based on SNARKs [GNS21]. When using $\mathbb{Z}_{2^k}$, this work obtains a designated-verifier SNARK, however, the scheme has not been implemented, and suffers from a dependency on expensive, public-key cryptography, as in many field-based SNARKs.

2.1.1 Contributions

In this work, we address the question of building efficient protocols for VOLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. Firstly, we show how to build a maliciously secure VOLE protocol over $\mathbb{Z}_{2^k}$, with efficiency comparable to state-of-the-art protocols over finite fields [BCG+19a, WYKW21a]. Our protocol introduces new consistency checks for verifying correctness of VOLE extension, which are tailored to overcome the difficulties of working with the ring $\mathbb{Z}_{2^k}$. Secondly, using our VOLE over $\mathbb{Z}_{2^k}$, we show how to adapt the QuickSilver protocol [YSWW21a] to the ring setting, obtaining an efficient ZK protocol called QuarkSilver that is dedicated to proving circuit satisfiability over $\mathbb{Z}_{2^k}$. Here,

we extend techniques from the MPC world [CDE+18] to be suitable for our ZK proof. Finally, we implemented and benchmarked both our VOLE and ZK protocols to demonstrate their performance. In a high-bandwidth, low-latency setting, our implementation achieves a throughput of 13–50 million VOLEs per second for 64 bit to 256 bit rings with 40 bit statistical security while transmitting only ≈ 1 bit per VOLE. Our QuarkSilver implementation is able to compute and verify 1.3 million 64 bit multiplications per second.

2.1.2 Our Techniques

Below, we expand on our contributions, the techniques involved and some more relevant background.

Challenge of Working in $\mathbb{Z}_{2^k}$.

Before delving into our protocols, we first briefly recap the main challenges when working with rings like $\mathbb{Z}_{2^k}$, compared with finite fields. When using VOLE for zero-knowledge, VOLE is used to *commit* the prover to its inputs and intermediate wire values in the circuit. This is possible by viewing each VOLE output $\mathsf{M}[x] = \Delta \cdot x + \mathsf{K}[x]$ as an information-theoretic homomorphic MAC in the input x.

When working over a finite field, it's easy to see that if a malicious prover can come up with a valid MAC $\mathsf{M}[\overline{x}]$ on an input $\overline{x} \neq x$, for the same key $\mathsf{K}[x]$, then the prover can recover the MAC key Δ from the relation:

$$\mathsf{M}[x] - \mathsf{M}[\overline{x}] = \Delta \cdot (x - \overline{x})$$

However, this relies on $x - \overline{x}$ being invertible, which is usually not the case when working over a ring such as $\mathbb{Z}_{2^k}$. Indeed, if $x - \overline{x} = 2^{k-1}$, then the prover can forge a MAC $\mathsf{M}[\overline{x}]$ with probability 1/2, since $\mathsf{M}[x] - \mathsf{M}[\overline{x}] \bmod 2^k$ now only depends on the least significant bit of Δ.

The SPD$\mathbb{Z}_{2^k}$ protocol [CDE+18] for multi-party computation showed how to work around this issue by extending the modulus to 2^{k+s}, for some statistical security parameter s. This way, it can be shown that the lower s bits of the key Δ are still enough to protect the integrity of the lower k bits of the message x.

Indeed, this was exactly the type of MAC scheme used in the recent work on conversions and ZK over rings [BBMH+21b]. However, as in the SPD$\mathbb{Z}_{2^k}$ protocols, further challenges arise when handling more complex protocols for verifying computation on MACed values.

Maliciously Secure VOLE Extension in $\mathbb{Z}_{2^k}$.

Current state-of-the-art VOLE protocols all stem from the approach of Boyle et al. [BCGI18], which builds a *pseudorandom correlation generator* based on (variants of) the *learning parity with noise* (LPN) assumption. This approach exploits the fact that sparse LPN errors can be used to compress secret-sharings of pseudorandom vectors, allowing the two

parties to generate a long, pseudorandom instance of a VOLE correlation in a succinct manner.

These protocols proceed by first constructing a protocol for *single-point* VOLE, where the sender's input vector has only a single non-zero entry. Then, the single-point VOLE protocol is repeated t times, to obtain a t-point VOLE where the sender's input is viewed as a long, sparse, LPN error vector. Finally, by combining t-point VOLE and the LPN assumption, the parties can locally transform this into pseudorandom VOLE by appling a linear mapping.

Using this blueprint leads to (random) VOLE protocols with communication much smaller than the output length. This can be seen as a form of *VOLE extension*, where in the first step, a small "seed" VOLE of length $m \ll n$ is used to create the single-point VOLEs, and then extended into a longer VOLE of length n. In the Wolverine protocol [WYKW21a], it was additionally observed that when repeating this process, it can greatly help communication if m of the n extended outputs are reserved and used to bootstrap the next iteration of the protocol, saving generation of fresh seed VOLEs.

With semi-honest security, the above approach can easily be instantiated over rings, following the protocols of [SGRR19, BCG+19a]. When adapting this protocol to malicious security, our main technical challenge is that previous works over fields [BCG+19a, WYKW21a] used a consistency check to verify correctness of the outputs, which involved taking random linear combinations over the field. Due to the existence of zero divisors, this technique does not directly translate to $\mathbb{Z}_{2^k}$. One possible approach, similarly to the MAC scheme described above, is to increase the size of the ring to, say, $\mathbb{Z}_{2^{k+s}}$, and use computations in the larger ring to ensure that the VOLEs are correct modulo 2^k. However, the problem is, it would then no longer be compatible with the bootstrapping technique of [WYKW21a]: to check consistency, the seed VOLE must be in the larger ring $\mathbb{Z}_{2^{k+s}}$, however, since the outputs are only in $\mathbb{Z}_{2^k}$, they can't then be used as a seed for the next execution! One solution would be to start with an even larger ring ($\mathbb{Z}_{2^{k+2s}}$), and keep decreasing the ring size after each iteration, but this would be far too expensive when done repeatedly.

Instead, we take a different approach. First, we adopt a hash-based check from [BCG+19a], which verifies correctness of a puncturable pseudorandom function based on a GGM tree, created during the protocol. This hash check (which we optimize by using universal hashing instead of a cryptographic hash function) works over rings as well as fields, however, it does not suffice to ensure consistency of the entire protocol. On top of this, we incorporate a linear combination check, however, one with binary coefficients instead of coefficients in the large ring. This type of check can be used over a ring, but allows a cheating prover to try to bypass the check and cheat successfully with probability 1/2. Nevertheless, we show that by allowing some additional leakage in the single-point VOLE functionality, we can still simulate the protocol with this check. For our final VOLE protocol, this leakage implies that a few noise coordinates of the LPN error vector may have leaked.

While previous protocols also allowed a limited form of leakage [BCG+19a, WYKW21a], in this case, ours is more serious since entire noise coordinates can be leaked with prob-

ability 1/2. To counter this, we analyze the state-of-the-art attacks on LPN, and show how to adjust the parameters and increase the noise rate accordingly.

Similarly to [WYKW21a], we focus on using the "primal" form of LPN, which was also used for semi-honest VOLE over $\mathbb{Z}_{2^k}$ in [SGRR19]. While the "dual" form of LPN, as considered in [BCGI18, BCG+19a, CRR21], achieves lower communication costs (and does not rely on bootstrapping), it involves a more costly matrix multiplication, which is expensive to implement. In [BCG+19a], dual-LPN was instantiated using quasi-cyclic codes to achieve $\tilde{O}(n)$ complexity, but this approach does not readily adapt to rings instead of fields; it is plausible that the fast, LDPC-based dual-LPN variant proposed in [CRR21] can be adapted to work over rings, but the security of this assumption has not been analyzed thoroughly.

Efficient Zero-Knowledge via QuarkSilver in $\mathbb{Z}_{2^k}$.

Given VOLE, the standard approach to obtaining a ZK proof is using the homomorphic MAC scheme described above. There, the prover first commits to the input $\mathbf{w}$ as well as all intermediate circuit wire values of $\mathcal{C}(\mathbf{w})$. Then, the prover must show consistency of all the wire values and that the output wire indeed contains 1. Since the MACs are linearly homomorphic, the main challenge is verifying multiplications. In QuickSilver [YSWW21a], to verify that committed values x, y, z satisfy $x \cdot y = z$, the parties locally compute a quadratic function on their MACs and MAC keys, obtaining a new value which has a consistent MAC only if the multiplication is correct.

The catch is that this new MAC relation being checked leads to a quadratic equation in the secret key Δ, instead of linear as before, which is chosen by a possibly dishonest prover. If this quadratic equation has a root in Δ, then the check passes. In the field case, this is not a problem as there are no more than two solutions to a quadratic equation, so we obtain a soundness error of $2/|\mathbb{F}|$. However, with rings, there can be many solutions. For instance, with

$$f(X) = aX^2 + bX + c \pmod{2^k},$$

if $a = 2^{k/2}$ and $b = c = 0$ then any multiple of $2^{k/4}$ is a possible choice for X, i.e. the check would erroneously pass for $2^{3k/4}$ choices of Δ. To remedy this, we reduce the number of valid solutions by working modulo 2^ℓ for some $\ell > k$, and adding the constraint on the solution that $\Delta \in \{0, \ldots, 2^s - 1\}$, where s is a statistical security parameter.

An additional challenge is that when checking a batch of multiplications, we actually check a random linear combination of a large number of these equations, which again leads to complications with zero divisors. By carefully analyzing the number of bounded solutions to equations of this type, and extending techniques from SPD$\mathbb{Z}_{2^k}$ [CDE+18] for handling linear combinations over rings, we show that it suffices to choose $\ell \approx k + 2(\sigma + \log \sigma)$ to achieve $2^{-\sigma}$ failure probability in the check. Overall, we obtain a communication complexity of ℓ bits per input and multiplication gate in the circuit.

2.2 Preliminaries

2.2.1 Notation

We use lower case, bold symbols for vectors $\mathbf{x}$ and upper case, bold symbols for matrices $\mathbf{A}$. We use κ as the computational and σ as the statistical security parameter. In our UC functionalities and proofs, $\mathcal{Z}$ denotes the environment, and $\mathcal{S}$ is the simulator, while $\mathcal{A}$ will refer to the adversary.

2.2.2 Vector OLE

Vector OLE (VOLE) is a two party functionality between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$ to obtain correlated random vectors of the following form: $\mathcal{S}$ obtains two vectors $\mathbf{u}, \mathbf{w}$, and $\mathcal{R}$ gets a random scalar Δ and a random vector $\mathbf{v}$ so that $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$ holds.

We parameterize the functionality with two values ℓ and s such that $s \leq \ell$. The scalar Δ is sampled from $\mathbb{Z}_{2^s}$, and the vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are sampled from $\mathbb{Z}_{2^\ell}^n$ where n denotes the size of the correlation. We require that the equation $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$ holds modulo 2^ℓ. The ideal functionality is described in Figure 2.1.

As in SPD$\mathbb{Z}_{2^k}$ [CDE+18], we can implement $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ using the oblivious transfer protocol (OT) of [Sch18]. Basing VOLE on OT has the drawback of quadratic communication costs in the ring size, since it requires one OT of size ℓ bit for each of the ℓ bits of a ring element. Hence, we would use this approach only once to create a set of base VOLEs. Then we can use the more efficient protocol presented in Section 2.4 to repeatedly generate large batches to VOLEs.

2.2.3 Equality Test

In our work, we use an equality test functionality $\mathcal{F}_{\mathsf{EQ}}$ (Figure 2.2) between two parties $\mathcal{P}, \mathcal{V}$ where $\mathcal{V}$ learns the input of $\mathcal{P}$. The equality check functionality can be implemented using a simple commit-and-open protocol, see e.g. [WYKW21a]. When using a hash function with 2κ bit output (modeled as random oracle) to implement the commitment scheme, the equality check of ℓ bit values can be implemented with $\ell + 3\kappa$ bit of communication.

2.2.4 Zero-Knowledge Proofs of Knowledge

In Figure 2.3 we provide an ideal functionality for zero-knowledge proofs. The functionality implies the standard definition of a ZKPoK as it is:

Complete because whenever $\mathcal{C}(\mathbf{w}) = 1$ then an honest verifier will accept.

Knowledge Sound because it outputs `true` iff $\mathcal{P}$ inputs, and thus knows, a satisfying assignment $\mathbf{w}$ for $\mathcal{C}$.

Zero-Knowledge because nothing beyond the check $\mathcal{C}(\mathbf{w}) \stackrel{?}{=} 1$ is leaked to $\mathcal{V}$.

VOLE for $\mathbb{Z}_{2^k}$: $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$

Let $\ell \geq s$.

Init This method is the first to be called by the parties. On input (Init) from both parties proceed as follows:

1. If $\mathcal{R}$ is honest, sample $\Delta \in_R \mathbb{Z}_{2^s}$ and send Δ to $\mathcal{R}$.
2. If $\mathcal{R}$ is corrupt, receive $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{S}$.
3. Δ is stored by the functionality.

All further (Init) queries are ignored.

Extend On input (Extend, n) from both parties proceed as follows:

1. If $\mathcal{R}$ is honest, sample $\mathbf{v} \in_R \mathbb{Z}^n_{2^\ell}$. Otherwise receive $\mathbf{v} \in_R \mathbb{Z}^n_{2^\ell}$ from $\mathcal{S}$.
2. If $\mathcal{S}$ is honest, sample $\mathbf{u} \in_R \mathbb{Z}^n_{2^\ell}$ and compute $\mathbf{w} := \Delta \cdot \mathbf{u} + \mathbf{v} \in \mathbb{Z}_{2^\ell}$. Otherwise receive $\mathbf{u} \in \mathbb{Z}^n_{2^\ell}$ and $\mathbf{w} \in \mathbb{Z}^n_{2^\ell}$ from $\mathcal{S}$ and then recompute $\mathbf{v} := \mathbf{w} - \Delta \cdot \mathbf{u} \in \mathbb{Z}^n_{2^\ell}$
3. Send $(\mathbf{u}, \mathbf{w})$ to $\mathcal{S}$ and $\mathbf{v}$ to $\mathcal{R}$.

Global-key Query If $\mathcal{S}$ is corrupted, receive (Guess, Δ') from $\mathcal{S}$ with $\Delta' \in \mathbb{Z}_{2^s}$. If $\Delta' = \Delta$, send success to $\mathcal{S}$ and ignore subsequent global-key queries. Otherwise, send abort to both parties and abort.

Figure 2.1: Ideal functionality VOLE over $\mathbb{Z}_{2^k}$.

Equality Test: $\mathcal{F}_{\mathsf{EQ}}$

On input $V_{\mathcal{P}}$ from $\mathcal{P}$ and $V_{\mathcal{V}}$ from $\mathcal{V}$:

1. Send $V_{\mathcal{P}}$ and $(V_{\mathcal{P}} \stackrel{?}{=} V_{\mathcal{V}})$ to $\mathcal{V}$.
2. If $\mathcal{V}$ is honest and $V_{\mathcal{P}} = V_{\mathcal{V}}$, or $\mathcal{V}$ is corrupted and sends continue, then send $(V_{\mathcal{P}} \stackrel{?}{=} V_{\mathcal{V}})$ to $\mathcal{P}$
3. If $\mathcal{V}$ is honest and $V_{\mathcal{P}} \neq V_{\mathcal{V}}$, or $\mathcal{V}$ is corrupted and sends abort, then send abort to $\mathcal{P}$.

Figure 2.2: Ideal functionality for equality tests.

Zero-Knowledge Functionality $\mathcal{F}_{\mathsf{ZK}}^{k}$

Prove: On input $(\mathsf{prove}, \mathcal{C}, \mathbf{w})$ from $\mathcal{P}$ and $(\mathsf{verify}, \mathcal{C})$ from $\mathcal{V}$ where $\mathcal{C}$ is a circuit over $\mathbb{Z}_{2^k}$ and $\mathbf{w} \in \mathbb{Z}_{2^k}^n$ for some $n \in \mathbb{N}$: Send true to $\mathcal{V}$ iff $\mathcal{C}(\mathbf{w}) = 1$, and false otherwise.

Figure 2.3: Ideal functionality for zero-knowledge proofs for circuit satisfiability.

2.2.5 The LPN Assumption Over Rings

The *Learning Parity with Noise* (LPN) assumption [BFKL94] states that, given the noisy dot product of many public vectors $\mathbf{a}_i$ with a secret vector $\mathbf{s}$, the result is indistinguishable from a vector of random values. Adding noise to indices is done by adding a noise vector $\mathbf{e}$ at the end, consisting of random values.

We rely on the following arithmetic variant of LPN over a ring $\mathbb{Z}_M$, as also considered in [BCGI18, SGRR19].

Definition 1 (LPN)**.** *Let $\mathcal{D}_{n,t}^M$ be a distribution over $\mathbb{Z}_M^n$ such that for any $t, n, M \in \mathbb{N}$, $\mathrm{Im}(\mathcal{D}_{n,t}^M) \in \mathbb{Z}_M^n$. Let G be a probabilistic code generation algorithm such that $\mathsf{G}(m, n, M)$ outputs a matrix $\mathbf{A} \in \mathbb{Z}_M^{m\times n}$. Let parameters m, n, t be implicit functions of security parameter κ. The $\mathsf{LPN}_{m,n,t,M}^{\mathsf{G}}$ assumptions states that:*

$$\{(\mathbf{A}, \mathbf{x}) \mid \mathbf{A} \leftarrow \mathsf{G}(m, n, M), \mathbf{s} \in_R \mathbb{Z}_M^m, \mathbf{e} \leftarrow \mathcal{D}_{n,t}^M, \mathbf{x} := \mathbf{s} \cdot \mathbf{A} + \mathbf{e}\}$$
$$\approx_C \{(\mathbf{A}, \mathbf{x}) \mid \mathbf{A} \leftarrow \mathsf{G}(m, n, M), \mathbf{x} \in_R \mathbb{Z}_M^n\}.$$

There exists two flavours of the LPN assumption; the primal (Definition 1) and the dual (Definition 2).

Definition 2 (Dual-LPN)**.** *Let $\mathcal{D}_{n,t}^M$ and G be defined as in Definition 1. Let $n', n \in \mathbb{N}$ be defined so that $n' > n$ and then let $\mathsf{G}^{\perp}$ be a probabilistic code generation algorithm such that $\mathsf{G}^{\perp}(n', n, M)$ outputs a matrix in $\{\mathbf{H} \in \mathbb{Z}_M^{n'\times n} \mid \mathrm{rank}(\mathbf{H}) = n \wedge \exists \mathbf{A} \in \mathrm{Im}(\mathsf{G}(n' - n, n', M))\,.\,\mathbf{A}\cdot\mathbf{H} = 0\}$. Let parameters n, n', t be implicit functions of security parameter κ. The $\mathsf{dual\text{-}LPN}_{n',n,t,M}^{\mathsf{G}}$ assumption states that:*

$$\{(\mathbf{H}, \mathbf{x}) \mid \mathbf{H} \leftarrow \mathsf{G}^{\perp}(n', n, M), \mathbf{e} \leftarrow \mathcal{D}_{n,t}^M, \mathbf{x} := \mathbf{e} \cdot \mathbf{H}\}$$
$$\approx_C \{(\mathbf{H}, \mathbf{x}) \mid \mathbf{H} \leftarrow \mathsf{G}^{\perp}(n', n, M), \mathbf{x} \in_R \mathbb{Z}_M^n\}.$$

Informally, the main advantage of the *primal* version of LPN is that there exist practical (and implemented) constructions of the LPN-friendly codes required for this. Specifically, one can choose the code matrix $\mathbf{A}$ from a family of codes G supporting *linear-time* matrix-vector multiplication, such as *d-local linear codes* so that each column of $\mathbf{A}$ has exactly d non-zero entries. According to [Ale03], the hardness of LPN for local linear codes is well-established. Its main disadvantage however, is that its output size can be

at most quadratic in the size of the seed, as intuitively, a higher stretch would make it significantly easier for an adversarial verifier to guess enough noiseless coordinates to allow efficient decoding via Gaussian Elimination [AG11].

The main advantage of the *dual* variant is that it allows for an arbitrary polynomial stretch. However, the compressive mapping used within the dual variant cannot have constant locality and is more challenging to instantiate. Recently, Silver [CRR21] proposed an instantiation of dual-LPN based on structured LDPC codes, which have been practically implemented over finite fields, and may plausibly also work over rings.

Attacks on LPN

We recall the main attacks on LPN, following the analysis of [EKM17, BCGI18, BCG+19a]. We refer to [EKM17] for a more thorough overview. Let $\mathcal{D}_{n,t}^{M}$ be the noise distribution with Hamming weight t. Recall that the LPN secret dimension is m, while the number of samples is n. We define the average noise rate to be $r = t/n$.

Pooled Gaussian Elimination This attack recovers $\mathbf{x}$ from $\mathbf{b} = \mathbf{x} \cdot \mathbf{A} + \mathbf{e}$ by guessing m non-noisy coordinates of $\mathbf{b}$, performs Gaussian elimination to find $\mathbf{x}$ and verifies that the guess of the m non-noisy coordinates was correct. [EKM17] introduced *Pooled* Gaussian elimination in order to reduce the samples required by regular Gaussian elimination. In Pooled Gaussian elimination, the adversary guesses m non-noise samples by picking them at random from a pool of fixed $N = m^2 \log^2 m$ LPN samples in each iteration, and then inverts the corresponding subsystem to get a potential solution x' and then checks if $x' = x$. For LPN with noise rate r, this attacks recovers the secret in time $\frac{m^3 \log^2 m}{(1-r)^m}$ using $m^2 \log^2 m$ samples.

Information Set Decoding (ISD) [Pra62] Breaking LPN is equivalent to breaking its dual variant, which may be interpreted as the task of decoding a random linear code from its syndrome. The best algorithms for this are improvements of Prange's ISD algorithm, which tries to find a size-t subset of the rows of $\mathbf{B}$ (the parity-check matrix of the code used within the Dual-LPN assumption) that spans $\mathbf{e} \cdot \mathbf{B}$.

The BKW Algorithm [BKW03] This is a variant of Gaussian elimination which achieves subexponential complexity, even for high-noise LPN. It requires a subexponential number of samples and can solve LPN over $\mathbb{F}_2$ in time $2^{O(m/\log(m/r))}$ using $2^{O(m/\log(m/r))}$ samples.

Combinations of the above [EKM17] The authors of [EKM17] conducted an extended study of the security of LPN and they described combinations and refinements of the previously mentioned attack (called *well-pooled Gauss attack*, *hybrid attack* and the *well-pooled MMT attack*). All of these attacks achieve subexponential time complexity, but require as many samples as their time complexity.

Scaled-down BKW [Lyu05] This is a variant of the BKW algorithm, tailored to LPN with polynomially-many samples. It solves the LPN problem in time $2^{O(m/\log\log(m/r))}$,

using $m^{1+\epsilon}$ samples (for any $\epsilon > 0$) and has worse performance in time and number of samples for larger fields.

Statistical Decoding [DAT17][1] The goal of all of the previous attacks is to recover the secret $\mathbf{x}$, whereas this attack directly attempts to distinguish $\mathbf{b} = \mathbf{x} \cdot \mathbf{A} + \mathbf{e}$ from random. By the singleton bound, the minimal distance of the dual code of $\mathbf{C}$ is at most $m+1$, hence there must be a parity-check equation for $\mathbf{C}$ of weight $m+1$, that is, a vector $\mathbf{v}$ such that $\mathbf{A} \cdot \mathbf{v}^T = 0$. Then, if $\mathbf{b}$ is random, $\mathbf{b} \cdot \mathbf{v}^T = 0$ with probability at most $1/|\mathbb{F}|$, whereas if $\mathbf{b}$ is a noisy encoding, it goes to zero with probability roughly $((n-m-1)/n)^{rn}$ [BCGI18].

Note that some of the above attacks are specialized to LPN over $\mathbb{F}_2$, while others work for more general fields. When working in $\mathbb{Z}_{2^\ell}$, though, one can always reduce the LPN instance modulo 2 and run the distinguisher for the problem in $\mathbb{F}_2$.

Dealing with Reduction Attacks Over Rings.

When working over a ring $\mathbb{Z}_M$ instead of a finite field, we must take care that the presence of zero divisors does not weaken security. For instance, a simple reduction attack was pointed out in [LWYY22], where noise values can become zero after reducing modulo a factor of M (for instance, in $\mathbb{Z}_{2^k}$, reducing the LPN sample modulo 2 cuts the number of noisy coordinates in half, significantly reducing security). To mitigate this attack, we always sample non-zero entries of the error vector $\mathbf{e}$ and matrix $\mathbf{A}$ to be in $\mathbb{Z}_M^*$, that is, invertible mod M.[2] While [LWYY22] did not consider the effect on the matrix $\mathbf{A}$, we observe that if $\mathbf{A}$ is sparse then its important to ensure that its sparsity cannot also be decreased through reduction.[3] With these countermeasures, we are not aware of any attacks on LPN in $\mathbb{Z}_M$ that perform better than the field case.

We elaborate below on our choice of primal-LPN distribution.

Choice of Matrix over $\mathbb{Z}_M$.

We choose a random, sparse matrix $\mathbf{A}$ with d non-zero entries per column. We choose each non-zero entry randomly from $\mathbb{Z}_M^*$, to ensure that it remains non-zero after reduction modulo any factor of M. We fix the sparsity to $d = 10$, as in previous works [BCGI18, SGRR19, WYKW21a], which according to [ADI+17, Zic17] suffices to ensure that $\mathbf{A}$ has a large dual distance, which implies the LPN samples are unbiased [CRR21].

[1] Statistical Decoding is also known as Low-Weight Parity Check [ADI+17, Zic17].

[2] This countermeasure was missing from the original version of this paper, before [LWYY22] was available.

[3] On the other hand, the LPN secret $\mathbf{s}$ must *not* be chosen over $\mathbb{Z}_M^*$, but instead uniformly over $\mathbb{Z}_M$, since if e.g. $\mathbf{s}$ was known to be odd over $\mathbb{Z}_{2^k}$ then solving the reduced instance modulo 2 would be trivial.

Noise Distribution in $\mathbb{Z}_M$.

The noise distribution $\mathcal{D}_{n,t}^M$ is chosen to have t expected non-zero coordinates. This can be done on expectation with a Bernoulli distribution, where each coordinate is either zero, or non-zero (and uniform otherwise) with probability t/n. In our applications, we instead use an exact noise weight, where $\mathcal{D}_{n,t}^M$ fixes t non-zero coordinates in the length-n vector.

Invertible Noise Terms. When working over a ring $\mathbb{Z}_M$, we sample the non-zero noise values to be in $\mathbb{Z}_M^*$, that is, invertible mod M. This prevents the reduction attack mentioned above, which would otherwise reduce the expected noise weight by a factor of two for $M = 2^k$.

Uniform vs Regular Noise Patterns. For fixed-weight noise, we speak of *two types* of error; *regular* or *uniform*. We call uniform errors the case where $\mathcal{D}_{n,t}^M$ is the uniform distribution over all weight-t vectors of $\mathbb{Z}_M^n$ with non-zero values in $\mathbb{Z}_M^*$. Implementing LPN-based PCGs with uniform errors has previously been investigated by [YWL+20, SGRR19]. It is commonly implemented by utilising a sub-protocol to place a single non-zero value within a vector of length $n' \ll n$ and then using Cuckoo hashing to generate a uniform distribution over n from several of these smaller vectors, ending up with the t points distributed randomly across the n coordinates.

Our construction uses a *regular* noise distribution for the primal-LPN instance. Here, the noise vector in $\mathbb{Z}_M^n$ is divided into t blocks of length $\lfloor n/t \rfloor$, such that each block has exactly one non-zero coordinate. Generally, using LPN with regular errors is practically more efficient than for uniform errors [YWL+20, WYKW21a].

2.3 Single-Point Vector OLE

Single-point VOLE is a specialized functionality that generates a VOLE correlation $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$ (see Section 2.2.2) where $\mathbf{u}$ has only one non-zero coordinate $\alpha \in [n]$. We consider a variant where u_α is not only non-zero, but additionally also required to be invertible.

We present an ideal functionality for single-point VOLE $\mathcal{F}_{\text{sp-vole2k}}^{\ell,s}$ in Figure 2.4. In the functionality, $\mathcal{S}$ obtains $\mathbf{u}, \mathbf{w} \in \mathbb{Z}_{2^\ell}^n \times \mathbb{Z}_{2^\ell}^n$, and $\mathcal{R}$ gets $\Delta, \mathbf{v} \in \mathbb{Z}_{2^s} \times \mathbb{Z}_{2^\ell}^n$. As in the full VOLE functionality $\mathcal{F}_{\text{vole2k}}^{\ell,s}$ we allow $\mathcal{S}$ to attempt to guess Δ. Additionally, $\mathcal{F}_{\text{sp-vole2k}}^{\ell,s}$ also allows $\mathcal{R}$ to obtain leakage on the non-zero index:

1. $\mathcal{R}$ is allowed to guess a set $I \subseteq [n]$ that should contain the index α. Upon correct guess, if $|I| = 1$ then it learns $\mathbf{u}_\alpha$ while if $|I| > 1$ the functionality continues. If $\alpha \notin I$ then the functionality aborts.

2. $\mathcal{R}$ is also allowed a second query for a set $J \subset [n]$ that might contain α where $|J| = n/2$. If $\mathcal{R}$ guesses correctly then the functionality outputs α, while it aborts otherwise.

Single-Point VOLE for $\mathbb{Z}_{2^\ell}$: $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$

This functionality extends the functionality $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$ (Figure 2.1). In addition to the methods (Init) and (Extend), it also provides the method (SP-Extend) and a modified global-key query.

SP-Extend On input (SP-Extend, n) with $n \in \mathbb{N}$ from both parties the functionality proceeds as follows:

1. Sample $\mathbf{u} \in_R \mathbb{Z}^n_{2^\ell}$ with a single entry invertible modulo 2^ℓ and zeros everywhere else, $\mathbf{v} \in_R \mathbb{Z}^n_{2^\ell}$, and compute $\mathbf{w} := \Delta \cdot \mathbf{u} + \mathbf{v} \in \mathbb{Z}^n_{2^\ell}$.
2. If $\mathcal{S}$ is corrupted, receive $\mathbf{u} \in \mathbb{Z}^n_{2^\ell}$ with at most one non-zero entry and $\mathbf{w} \in \mathbb{Z}^n_{2^\ell}$ from $\mathcal{S}$, and recompute $\mathbf{v} := \mathbf{w} - \Delta \cdot \mathbf{u}$.
3. If $\mathcal{R}$ is corrupted:
   (a) Receive a set $I \subseteq [n]$ from $\mathcal{S}$. Let $\alpha \in [n]$ be the index of the non-zero entry $\mathbf{u}$, and let $\beta := u_\alpha$. If $I = \{\alpha\}$, then send (success, β) to $\mathcal{R}$. If $\alpha \in I$ and $|I| > 1$, then send success to $\mathcal{R}$ and continue. Otherwise send abort to both parties and abort.
   (b) Receive either (continue) or (query, J) from $\mathcal{S}$. If (continue) was received, continue with Step 3c. If (query, J) with $J \subset [n]$ and $|J| = \frac{n}{2}$ was received and $\alpha \in J$, then send α to $\mathcal{S}$. Otherwise, send abort to all parties, and abort.
   (c) Receive $\mathbf{v} \in \mathbb{Z}^n_{2^\ell}$ from $\mathcal{S}$, and recompute $\mathbf{w} := \Delta \cdot \mathbf{u} + \mathbf{v}$.
4. Send $(\mathbf{u}, \mathbf{w})$ to $\mathcal{S}$ and $\mathbf{v}$ to $\mathcal{R}$.

Global-key Query If $\mathcal{S}$ is corrupted, receive (Guess, Δ', s') from $\mathcal{S}$ with $s' \leq s$ and $\Delta' \in \mathbb{Z}_{2^{s'}}$. If $\Delta' = \Delta \pmod{2^{s'}}$, send success to $\mathcal{S}$. Otherwise, send abort to both parties and abort.

Figure 2.4: Ideal functionality for a leaky single-point VOLE.

The leakage is somewhat inherent to our protocol which we use to realize $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$.

Protocol Overview.

Our protocol $\Pi^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ (Figure 2.5) achieves active security using consistency checks inspired by the constructions from [BCG+19a] and [WYKW21a]. We now give a high-level overview.

As a setup, we assume functionalities $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{EQ}}$. For $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ we assume that $\mathcal{R}$ called (Init) already, thus setting Δ. Additionally, we require two pseudorandom generators (PRGs; with certain extra properties that we clarify in Section 2.3.1) to create a GGM tree. Recall, the GGM construction [GGM84] builds a PRF from a length-doubling PRG, by recursively expanding a PRG seed into 2 seeds, defining a complete binary tree where each of the n leaves is one evaluation of the PRF. We use this to build a puncturable PRF, where a subset of intermediate tree nodes is given out, enabling evaluating the PRF at all-but-one of the points in the domain.

The sender $\mathcal{S}$ begins by picking a random index α from $[n]$, and β randomly from $\mathbb{Z}^*_{2^\ell}$. This defines the vector $\mathbf{u}$ where $\mathbf{u}_\alpha = \beta$ and every other index is 0. $\mathcal{S}$ and $\mathcal{R}$ use a single VOLE from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ to authenticate β, resulting in the receiver holding γ and the sender holding δ, β such that $\delta = \Delta \cdot \beta + \gamma$.

To extend this correlation to the whole vector $\mathbf{u}$, $\mathcal{R}$ computes a GGM tree with $2n$ leaves. We consider all n leaves that are "left children" of their parent as comprising the vector $\mathbf{v}$. Using $\log_2(n)$ instances of $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{S}$ learns all "right children" as well as all of the "left children" except the one at position α – meaning that the sender learns $\mathbf{v}$ for all indices except α. $\mathcal{S}$ now sets $\mathbf{w}_i = \mathbf{v}_i$ for $i \neq \alpha$. This gives a valid correlation on these $n-1$ positions, because since $\mathbf{u}_i = 0$ for $i \neq \alpha$, we have that $\mathbf{w}_i = \Delta \cdot \mathbf{u}_i + \mathbf{v}_i$.

What remains in the protocol is for $\mathcal{S}$ to learn $\mathbf{w}_\alpha = \Delta \cdot \mathbf{u}_\alpha + \mathbf{v}_\alpha$ without revealing α and β to $\mathcal{R}$. Using the output of the VOLE instance, if $\mathcal{R}$ computes $d \leftarrow \gamma - \sum_{j=1}^{n} \mathbf{v}_j$ and sends d to $\mathcal{S}$, then $\mathcal{S}$ can compute

$$\begin{aligned}
\mathbf{w}_\alpha &= \delta - d - \sum_{j \in [n] \setminus \{\alpha\}} \mathbf{w}_j \\
&= \delta - \Big(\gamma - \sum_{i \in [n]} \mathbf{v}_i\Big) - \sum_{j \in [n] \setminus \{\alpha\}} \mathbf{w}_j \\
&= \delta - \gamma + \mathbf{v}_\alpha = \Delta \cdot \beta + \mathbf{v}_\alpha
\end{aligned}$$

which is exactly the missing value for the correlation. While this protocol can somewhat easily be proven secure against a dishonest $\mathcal{S}$ (assuming that the hybrid functionalities are actively secure), a corrupted $\mathcal{R}$ can cheat in two ways:

1. It can provide inconsistent GGM tree values to the $\mathcal{F}_{\mathsf{OT}}$ instances, thus leading to unpredictable protocol behavior.

2. It can construct d incorrectly.

To ensure a "somewhat consistent" GGM tree (and inputs to $\mathcal{F}_{\mathsf{OT}}$) we use a check that sacrifices all the leaves that are "right children". Here, $\mathcal{R}$ has to send a random linear combination of these, over a binary extension field, with $\mathcal{S}$ choosing the coefficients. The check makes sure that if it passes, then the "left children" are consistent for every choice of α that would have made $\mathcal{S}$ not abort. This reduces arbitrary leakage to an essentially unavoidable selective failure attack (due to the use of $\mathcal{F}_{\mathsf{OT}}$).

To prevent the second attack, the sender and receiver use an additional VOLE from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and perform a random linear combination check to ensure correctness of the value d. Due to the binary coefficients used in the linear combination over $\mathbb{Z}_{2^\ell}$, our check only has soundness $1/2$. This, however, suffices to prove security if we relax the functionality by allowing a corrupt receiver to learn α with probability $1/2$. This way, in the simulation in our security proof, if the challenge vector $\boldsymbol{\chi}$ is such that the receiver passes the check despite cheating, the simulator can still extract a valid input using its knowledge of α.

The full protocol is presented in Figure 2.5. Before proving security of it, we first recap the Puncturable PRF from GGM construction and its security properties.

2.3.1 Checking Consistency of the GGM Construction

We use the GGM [GGM86] construction to implement a puncturable PRF F with domain $[n]$ and range $\{0,1\}^\kappa$.

In a puncturable PRF (PPRF), one party P_1 generates a PRF key k, and then both parties engage in a protocol where the second party P_2 obtains a punctured key $k\{\alpha\}$ for an index $\alpha \in [n]$ of its choice. With $k\{\alpha\}$, it is possible for P_2 to evaluate F at all points $[n] \setminus \{\alpha\}$ so that $F(k,i) = F(k\{\alpha\}, i)$ for $i \neq \alpha$, while nothing about $F(k,\alpha)$ is revealed. More formally:

Definition 3 (Adapted from [BCG+19a])**.** *A puncturable pseudorandom function (PPRF) with keyspace $\mathcal{K}$, domain $[n]$ and range $\{0,1\}^\kappa$ is a pseudorandom function F with an additional keyspace $\mathcal{K}_p$ and 3 PPT algorithms* KeyGen*,* Gen*,* $\mathsf{PuncEval}$ *such that*

KeyGen *on input 1^κ outputs a random key $k \in \mathcal{K}$.*

Gen *on input n, k outputs $\{F(k,i), k\{i\}\}_{i\in[n]}$ where $k\{i\} \in \mathcal{K}_p$.*

PuncEval *on input $n, \alpha, k\{\alpha\}$ outputs $\mathbf{v}^\alpha$ such that $\mathbf{v}^\alpha \in (\{0,1\}^\kappa)^n$.*

where $F(k,i) = \mathbf{v}^\alpha_i$ for all $i \neq \alpha$ and no PPT adversary $\mathcal{A}$, given $n, \alpha, k\{\alpha\}$ as input, can distinguish $F(k,\alpha)$ from a uniformly random value in $\{0,1\}^\kappa$ except with probability $\mathsf{negl}(\kappa)$*.*

For simplicity, we describe the algorithms for domains of size $n = 2^h$ for some $h \in \mathbb{N}$. By pruning the tree appropriately, the procedures can be adapted to support domain sizes that are not powers of two. Throughout the coming sections, we let $\alpha_1, \ldots, \alpha_h$ be the bit decomposition of $\alpha = \sum_{i=0}^{h-1} 2^i \cdot \alpha_{h-i}$, and let $\overline{\alpha}_i$ denote the complement. Let κ

Single-Point VOLE for $\mathbb{Z}_{2^\ell}$: $\Pi^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$

For the (Init) and (Extend) operations, the parties simply query $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$.

SP-Extend For (SP-Extend, n): Let $h := \lceil \log n \rceil$ and $\sigma' := \sigma + 2h$.

1. The parties send (Extend, 1) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. $\mathcal{S}$ receives $a, c \in \mathbb{Z}_{2^\ell}$ and $\mathcal{R}$ receives $b \in \mathbb{Z}_{2^\ell}$ such that $c = \Delta \cdot a + b \pmod{2^\ell}$ holds.
2. $\mathcal{S}$ samples $\alpha \in_R [n], \beta \in_R \mathbb{Z}^*_{2^\ell}$ and lets $\mathbf{u} \in \mathbb{Z}^n_{2^\ell}$ be the vector with $u_\alpha = \beta$ and $u_i = 0$ for all $i \neq \alpha$.
3. $\mathcal{S}$ sets $\delta := c$ and sends $a' := \beta - a \in \mathbb{Z}_{2^\ell}$ to $\mathcal{R}$. $\mathcal{R}$ computes $\gamma := b - \Delta \cdot a' \in \mathbb{Z}_{2^\ell}$. Now, $\delta = \Delta \cdot \beta + \gamma \pmod{2^\ell}$.
4. $\mathcal{R}$ computes $k \leftarrow \mathsf{GGM.KeyGen}(1^\kappa)$, runs $(\mathbf{v}, \mathbf{t}, (\overline{K}^i_0, \overline{K}^i_1)_{i\in[h]}, \overline{K}^{h+1}_1) \leftarrow \mathsf{GGM.Gen}(n, k)$, and sends $\overline{K}^{h+1} := \overline{K}^{h+1}_1 \in \mathbb{F}_{2^{\sigma'}}$ to $\mathcal{S}$.
5. Write $\alpha = \sum_{i=0}^{h-1} 2^i \cdot \alpha_{h-i}$, for $\alpha_i \in \{0,1\}$. For $i \in [h]$, the parties call $\mathcal{F}_{\mathsf{OT}}$ where $\mathcal{S}$, acting as the receiver, inputs $\overline{\alpha}_i$ and $\mathcal{R}$ inputs $(\overline{K}^i_0, \overline{K}^i_1)_{i\in[h]}$ to $\mathcal{F}_{\mathsf{OT}}$. $\mathcal{S}$ receives $\overline{K}^i := \overline{K}^i_{\overline{\alpha}_i}$.
6. *Check the GGM tree:*
   (a) $\mathcal{S}$ samples $\boldsymbol{\xi} \in_R \mathbb{F}^n_{2^{\sigma'}}$ and sends $\boldsymbol{\xi}$ to $\mathcal{R}$.[a]
   (b) $\mathcal{R}$ computes $\Gamma := \langle \boldsymbol{\xi}, \mathbf{t} \rangle \in \mathbb{F}_{2^{\sigma'}}$ and sends Γ to $\mathcal{S}$.
   (c) $\mathcal{S}$ runs $\mathbf{v}^\alpha \leftarrow \mathsf{GGM.PuncEval}(n, \alpha, (\overline{K}^i)_{i\in[h+1]})$ followed by $\mathsf{GGM.Check}(n, \alpha, (\overline{K}^i)_{i\in[h+1]}, \boldsymbol{\xi}, \Gamma)$. If the latter returns $\perp$, $\mathcal{S}$ aborts. Otherwise it has obtained $(v_j)_{j\in[n]\setminus\{\alpha\}}$.
7. $\mathcal{R}$ sends $d := \gamma - \sum_{j=1}^n v_j \in \mathbb{Z}_{2^\ell}$ to $\mathcal{S}$. $\mathcal{S}$ defines $\mathbf{w} \in \mathbb{Z}^n_{2^\ell}$ such that $w_j := v_j$ for $j \in [n] \setminus \{\alpha\}$ and $w_\alpha := \delta - d - \sum_{1\le j\le n, j\neq\alpha} w_j$. Then $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$.
8. *Check consistency of d:*
   (a) The parties send (Extend, 1) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. $\mathcal{S}$ receives $x, z \in \mathbb{Z}_{2^\ell}$ and $\mathcal{R}$ receives $y^* \in \mathbb{Z}_{2^\ell}$ such that $z = \Delta \cdot x + y^* \pmod{2^\ell}$ holds.
   (b) $\mathcal{S}$ samples $\boldsymbol{\chi} \in_R \{0,1\}^n$ with $\mathrm{HW}(\boldsymbol{\chi}) = \frac{n}{2}$ and sends it to $\mathcal{R}$.[b]
   (c) $\mathcal{S}$ computes $x^* := \chi_\alpha \cdot \beta - x \in \mathbb{Z}_{2^\ell}$ and sends x^* to $\mathcal{R}$. $\mathcal{R}$ computes $y := y^* - \Delta \cdot x^* \in \mathbb{Z}_{2^\ell}$. Then $z = y + \Delta \cdot \chi_\alpha \cdot \beta$.
   (d) $\mathcal{S}$ computes $V_\mathcal{S} := \sum_{i=1}^n \chi_i \cdot w_i - z$, and $\mathcal{R}$ computes $V_\mathcal{R} := \sum_{i=1}^n \chi_i \cdot v_i - y$. They send $V_\mathcal{S}, V_\mathcal{R}$ to $\mathcal{F}_{\mathsf{EQ}}$. If it returns (abort), then abort.
9. $\mathcal{S}$ outputs $(\mathbf{u}, \mathbf{w})$, and $\mathcal{R}$ outputs $\mathbf{v}$.

[a] Instead of sending the whole vector $\boldsymbol{\xi}$, $\mathcal{S}$ can send a κ bit random seed which is then expanded with a PRG to obtain $\boldsymbol{\xi}$.

[b] Again, $\mathcal{S}$ can send a short seed instead of $\boldsymbol{\chi}$.

Figure 2.5: Protocol instantiating $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ in the $(\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model.

be a computational and σ be a statistical security parameter. Define $\sigma' := \sigma + 2\log n$ and let $\mathsf{G}\colon \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$ and $\mathsf{G}'\colon \{0,1\}^\kappa \to \mathbb{Z}_{2^\ell} \times \mathbb{F}_{2^{\sigma'}}$ be two PRGs.

Recall that to achieve malicious security when generating a PPRF key in our protocol, we use the redundancy introduced from extending the domain to size $2n$, and check consistency by letting the receiver provide a hash of all the right leaves of the GGM tree. In order for the right leaves of the GGM tree to fix a unique tree, we require the PRG used for the final layer $\mathsf{G}'\colon \{0,1\}^\kappa \to \mathbb{Z}_{2^\ell} \times \mathbb{F}_{2^{\sigma'}}$ to satisfy the *right-half injectivity* property[4] as defined below.

Definition 4. *We say that a function $f = (f_0, f_1)\colon \{0,1\}^\kappa \to \mathbb{Z}_{2^\ell} \times \mathbb{F}_{2^{\sigma'}}, x \mapsto (f_0(x), f_1(x))$ is* right-half injective*, if its restriction to the right-half of the output space $f_1\colon \{0,1\}^\kappa \to \mathbb{F}_{2^{\sigma'}}$ is injective.*

In order to achieve active security of our construction, we provide an additional algorithm Check, together with a finite challenge set Ξ. This algorithm, on input $n, \alpha, k\{\alpha\}$, a challenge ξ and a checking value Γ outputs $\top$ or $\bot$.

Definition 5 (PPRF consistency)**.** *Let F be a PPRF and let Ξ be a challenge set whose size depends on a statistical security parameter σ. Consider the following game for* Check*:*

1. $(k\{1\}, \ldots, k\{n\}, \mathsf{state}) \leftarrow \mathcal{A}(1^\kappa, n)$.
2. $\xi \in_R \Xi$
3. $\Gamma \leftarrow \mathcal{A}(1^\kappa, \mathsf{state}, \xi)$
4. *For all $\alpha \in [n]$, let $\mathbf{v}^\alpha \leftarrow \mathsf{PuncEval}(1^\kappa, \alpha, k\{\alpha\})$.*
5. *Define $I := \{\alpha \in [n] \mid \top = \mathsf{Check}(n, \alpha, k\{\alpha\}, \xi, \Gamma)\}$.*
6. *We say $\mathcal{A}$ wins the game if there exists $\alpha \neq \alpha' \in I$ such that there is an index $i \in [n] \setminus \{\alpha, \alpha'\}$ with $v_i^\alpha \neq v_i^{\alpha'}$.*

We say that F has consistency *if no algorithm $\mathcal{A}$ wins the above game with probability more than $2^{-\sigma}$.*

Our algorithms $\mathsf{GGM.KeyGen}$, $\mathsf{GGM.Gen}$, $\mathsf{GGM.PuncEval}$, $\mathsf{GGM.Check}$, which are used to generate the key, set up the punctured keys, evaluate and check consistency of the punctured keys in our protocol are then as follows:

1. $\mathsf{GGM.KeyGen}(1^\kappa)$ samples $k \in \{0,1\}^\kappa$ uniformly at random and outputs it.
2. $\mathsf{GGM.Gen}(n, k)$ where $n = 2^h$ and $k \in \{0,1\}^\kappa$ is a key:
   (a) Set $K_0^0 \leftarrow k$.

[4] As noted in [BCG+19a], this can be replaced with a weaker notion of right-half collision resistance, which is easier to achieve in practice.

(b) For each level $i \in [h]$, and for $j \in \{0, \dots, 2^{i-1} - 1\}$ compute $(K^i_{2j}, K^i_{2j+1}) \leftarrow \mathsf{G}(K^{i-1}_j)$.

(c) For $i \in [h]$, set $\overline{K}^i_0 \leftarrow \bigoplus_{j=0}^{2^{i-1}-1} K^i_{2j}$ and $\overline{K}^i_1 \leftarrow \bigoplus_{j=0}^{2^{i-1}-1} K^i_{2j+1}$.

(d) For $j \in [2^h]$ compute $v_j, t_j \leftarrow \mathsf{G}'(K^h_{j-1})$, and set $\mathbf{v} := (v_1, \dots, v_{2^h})$ and $\mathbf{t} := (t_1, \dots, t_{2^h})$.

(e) Compute $\overline{K}^{h+1}_1 \leftarrow \sum_{j \in [2^h]} t_i$.

(f) Output $(\mathbf{v}, \mathbf{t}, (\overline{K}^i_0, \overline{K}^i_1)_{i \in [h]}, \overline{K}^{h+1}_1)$.

3. $\mathsf{GGM.PuncEval}(n, \alpha, (\overline{K}^i)_{i \in [h+1]})$ where $n = 2^h$, $\alpha \in [n]$, and $\overline{K}^i \in \{0,1\}^\kappa$:

(a) Set $K^1_{\overline{\alpha}_1} \leftarrow \overline{K}^1$.

(b) For each level $i \in \{2, \dots, h\}$:

i. Let $x := \sum_{j=1}^{i-1} 2^{j-1} \cdot \alpha_{i-j}$

ii. For $j \in \{0, \dots, 2^{i-1} - 1\} \setminus \{x\}$, compute $(K^i_{2j}, K^i_{2j+1}) \leftarrow \mathsf{G}(K^{i-1}_j)$.

iii. Compute $K^i_{2x+\overline{\alpha}_i} \leftarrow \overline{K}^i \oplus \bigoplus_{\substack{0 \le j < 2^{i-1} \\ j \ne x}} K^i_{2j+\overline{\alpha}_i}$.

(c) For the last level $h+1$:

i. For $j \in [2^h] \setminus \{\alpha\}$ compute $(v_j, t_j) \leftarrow \mathsf{G}'(K^h_{j-1})$

(d) Output $(v_j)_{j \in [2^h] \setminus \{\alpha\}}$.

4. $\mathsf{GGM.Check}(n, \alpha, (\overline{K}^i)_{i \in [h+1]}, (\xi_i)_{i \in [n]}, \Gamma)$ where $n = 2^h$, and $\overline{K}^i \in \{0,1\}^\kappa$, $\xi_i \in \mathbb{F}_{2^{\sigma'}}$, and $\Gamma \in \mathbb{F}_{2^{\sigma'}}$:

(a) For $j \in [2^h] \setminus \{\alpha\}$ recompute t_j as in $\mathsf{GGM.PuncEval}$.

(b) Compute $t_\alpha \leftarrow \overline{K}^{h+1} - \sum_{j \in [2^h] \setminus \{\alpha\}} t_j$.

(c) If $\Gamma = \sum_{i \in [n]} \xi_i \cdot t_i$, output $\top$. Otherwise, output $\bot$.

In comparison to Definition 3 $\mathsf{GGM.Gen}$ computes a compressed version of all keys. The pseudorandomness for GGM, as defined in Definition 3, follows from the standard pseudorandomness argument of the GGM construction [KPTZ13, BW13, BGI14].

The following theorem shows that the check ensures that a corrupted P_1 cannot create an inconsistent GGM tree, where P_2 obtains different values depending on α.

Theorem 2 (Consistency of the GGM Tree)**.** *Let $n = 2^h \in \mathbb{N}$, $\sigma' = \sigma + 2h$, and G, G' as above, and let $\mathcal{A}$ be any time adversary. If G' is right-half injective, then $\mathcal{A}$ can win the game in Definition 5 with probability at most $2^{-(\sigma+1)}$.*

To give the proof, we first define two lemmata which will simplify the proof.

Lemma 2. *For $n, \sigma \in \mathbb{N}$, the set $\mathcal{H}^n_\sigma := \{\mathbf{z} \mapsto \langle \mathbf{z}, \boldsymbol{\xi} \rangle \mid \xi \in \mathbb{F}^n_{2^\sigma}\}$ of functions $\mathbb{F}^n_{2^\sigma} \to \mathbb{F}_{2^\sigma}$ is a universal family of hash functions, i.e., for any two $\mathbf{x} \ne \mathbf{y} \in \mathbb{F}^n_{2^\sigma}$, we have $\Pr_{h \in_R \mathcal{H}^n_\sigma}[h(\mathbf{x}) = h(\mathbf{y})] = 2^{-\sigma}$.*

Proof. Halevi and Krawczyk [HK97] give the proof for Δ-universality for prime fields, but it works in the same way for normal universality and other finite fields: Let $\mathbf{x} \neq \mathbf{y} \in \mathbb{F}_{2^\sigma}^n$, and let $\mathbf{z}_h$ denote the vector corresponding to $h \in \mathcal{H}_\sigma^n$. W.l.o.g. assume $x_1 \neq y_1$. Then

$$\Pr_{h \in_R \mathcal{H}_\sigma^n}[h(\mathbf{x}) = h(\mathbf{y})] = \Pr_{h \in_R \mathcal{H}_\sigma^n}[\langle \mathbf{z}_h, \mathbf{x} \rangle = \langle \mathbf{z}_h, \mathbf{y} \rangle]$$
$$= \Pr_{h \in_R \mathcal{H}_\sigma^n}\left[z_{h,1} \cdot (x_1 - y_1) = -\sum_{i=2}^{n} z_{h,i} \cdot (x_i - y_i)\right] = 2^{-\sigma},$$

since $z_{h,1} \cdot (x_1 - y_1)$ is uniform in $\mathbb{F}_{2^\sigma}$ and independent of the right-hand side. □

Lemma 3. *For $n, \sigma \in \mathbb{N}$, and any $\mathbf{z}_1, \ldots, \mathbf{z}_n \in \mathbb{F}_{2^\sigma}$, the collision probability is*

$$\Pr_{h \in_R \mathcal{H}_\sigma^n}\left[\exists i, j \in [n] \,.\, \mathbf{z}_i \neq \mathbf{z}_j \wedge h(\mathbf{z}_i) = h(\mathbf{z}_j)\right] \leq \frac{n(n-1)}{2} \cdot 2^{-\sigma} \leq 2^{-(\sigma - 2\log(n) + 1)}.$$

Proof. We first apply the union bound and then use that $\mathcal{H}_\sigma^n$ is a universal family of hash functions:

$$\Pr_{h \in_R \mathcal{H}_\sigma^n}[\exists i, j \in [n] \,.\, \mathbf{z}_i \neq \mathbf{z}_j \wedge h(\mathbf{z}_i) = h(\mathbf{z}_j)]$$
$$= \Pr_{h \in_R \mathcal{H}_\sigma^n}\left[\bigvee_{i,j \in [n]} \mathbf{z}_i \neq \mathbf{z}_j \wedge h(\mathbf{z}_i) = h(\mathbf{z}_j)\right]$$
$$\leq \sum_{1 \leq i < j \leq n} \Pr_{h \in_R \mathcal{H}_\sigma^n}[\mathbf{z}_i \neq \mathbf{z}_j \wedge h(\mathbf{z}_i) = h(\mathbf{z}_j)] = \frac{n(n-1)}{2} \cdot 2^{-\sigma}$$
$$\leq \frac{n^2}{2} \cdot 2^{-\sigma} = \frac{(2^{\log n})^2}{2} \cdot 2^{-\sigma} = 2^{2\log(n) - 1} \cdot 2^{-\sigma} = 2^{-(\sigma - 2\log(n) + 1)}$$ □

Proof of Theorem 2. If $\mathcal{A}$ wins, then there exists indices $\alpha \neq \alpha' \in I$ such that $\mathbf{v}^\alpha$ and $\mathbf{v}^{\alpha'}$ are not consistent. So we have an index $i \in [n] \setminus \{\alpha, \alpha'\}$ with $v_i^\alpha \neq v_i^{\alpha'}$. The values v_i^α and $v_i^{\alpha'}$ were derived from the keys $K_i^{h,\alpha}$ and $K_i^{h,\alpha'}$ using G':

$$\mathsf{G}'(K_i^{h,\alpha}) = (v_i^\alpha, c_i^\alpha) \qquad \mathsf{G}'(K_i^{h,\alpha'}) = (v_i^{\alpha'}, c_i^{\alpha'})$$

Since $v_i^\alpha \neq v_i^{\alpha'}$, we have $K_i^{h,\alpha} \neq K_i^{h,\alpha'}$. Due to the right-half injectivity of G' it follows $c_i^\alpha \neq c_i^{\alpha'}$ and, thus, also $\mathbf{t}^\alpha \neq \mathbf{t}^{\alpha'}$. Finally, $\langle \boldsymbol{\xi}, \mathbf{t}^\alpha \rangle = \langle \boldsymbol{\xi}, \mathbf{t}^{\alpha'} \rangle = \Gamma$ must hold.

By Lemma 2, the function $h(\mathbf{t}) := \langle \boldsymbol{\xi}, \mathbf{t} \rangle$ is a universal hash function sampled uniformly from the family $\{\mathbf{t} \mapsto \langle \boldsymbol{\xi}, \mathbf{t} \rangle \mid \boldsymbol{\xi} \in \mathbb{F}_{2^{\sigma'}}^n\}$. Note that Step 4 is deterministic and the $(\mathbf{v}^\alpha, \mathbf{t}^\alpha)$ depend only on the values that $\mathcal{A}$ produced in Step 1. This implies that $\boldsymbol{\xi}$ is independent of the $\mathbf{t}^\alpha$. So the adversary can only win the game if there is a collision among the $\mathbf{t}^\alpha$ under the randomly sampled hash function h. By Lemma 3, we can bound this probability by $2^{-(\sigma' - 2h + 1)} = 2^{-(\sigma+1)}$. □

2.3.2 Security of $\Pi^{\ell,s}_{\textsf{sp-vole2k}}$

Theorem 3. *The protocol $\Pi^{\ell,s}_{\textsf{sp-vole2k}}$ (Figure 2.5) securely realizes the functionality $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ in the $(\mathcal{F}^{\ell,s}_{\textsf{vole2k}}, \mathcal{F}_{\textsf{OT}}, \mathcal{F}_{\textsf{EQ}})$-hybrid model: No PPT environment $\mathcal{Z}$ can distinguish the real execution of the protocol from a simulated one except with probability $2^{-(\sigma+1)} + \textsf{negl}(\kappa)$.*

Due to the complexity of the proof, we give a brief intuition on how it goes.

In the proof, we construct simulators for a corrupted sender and receiver. For the corrupted sender, the simulator follows the protocol by behaving like an honest receiver, but additionally extracts α from the interactions of the dishonest sender with $\mathcal{F}_{\textsf{OT}}$ and β from the VOLE. Its choice of GGM tree as well as other messages are used to define a consistent vector $\mathbf{w}$ that it sends to the functionality. A subtlety here is simulating the equality check in Step 8d of the protocol, as a corrupt sender can pass this with an ill-formed x^* if it can guess a portion of Δ used in the VOLE-functionality correctly. The simulator must make a key query to $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ to simulate the success event correctly. Another issue is that d sent by an honest receiver has a different distribution than how it is chosen in the simulation, but we show that any distinguisher can break the pseudorandomness of the GGM PPRF.

In the simulation for the corrupted receiver, the simulator first translates $\mathcal{F}_{\textsf{OT}}$ inputs into leakage queries to the functionality. For this, we know that due to Step 6c any adversarial choice leads to consistent GGM tree leaves, so the simulator chooses the set of indices where the check in this Step would pass as leakage input to the functionality $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$. This query then allows the simulator to create a valid transcript: if the attacker guessed α exactly correct (the set is of size 1), then the simulator obtains β from the functionality and can directly follow the protocol with the honest inputs. If the adversary instead guessed a set of size > 1 correctly that contains the secret α, then the simulator can reconstruct the whole GGM tree and thus a potential input $\mathbf{v}$. This furthermore allows the simulator to detect an inconsistent d that is sent by the corrupt receiver. An inconsistent d can be shown to translate into a selective failure attack on the equality check in Step 8d of the protocol, which requires the simulator to make the second leakage query. If it succeeds, then it obtains α and can adjust $\mathbf{v}_\alpha$ accordingly. Below, we formally give the proof of Theorem 3.

Proof of Theorem 3. First we cover the case of a corrupted sender, then that of a corrupted receiver.

Malicious Sender. The simulation is setup as follows: $\mathcal{S}$ simulates a party $\mathcal{S}^*$ in its head and gives control to $\mathcal{Z}$, and sends $(\textsf{corrupt}, \mathcal{S})$ to $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$. It also simulates instances of $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$, $\mathcal{F}_{\textsf{OT}}$, and $\mathcal{F}_{\textsf{EQ}}$. Since the calls to $(\textsf{Init})$ and $(\textsf{Extend})$ are simply forwarded to $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$, $\mathcal{S}$ can just simulate the interaction with the ideal functionality. The main part of proof is the simulation of $(\textsf{SP-Extend}, n)$. Let $h := \lceil \log n \rceil$.

1. $\mathcal{S}$ simulates the call $(\textsf{Extend}, 1)$ to $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$ and receives $a, c \in \mathbb{Z}_{2^\ell}$ from $\mathcal{S}^*$.
2. Receive $a' \in \mathbb{Z}_{2^\ell}$ from $\mathcal{S}^*$. Compute $\beta := a' + a$ and set $\delta := c$.

3. Compute $k \leftarrow \mathsf{GGM.KeyGen}(1^\kappa)$ and execute $(\mathbf{v}, \mathbf{t}, (\overline{K}_0^i, \overline{K}_1^i)_{i\in[h]}, \overline{K}_1^{h+1}) \leftarrow \mathsf{GGM.Gen}(n, k)$.
4. Send $\overline{K}_1^{h+1}$ to $\mathcal{S}^*$.
5. Simulate invocation of $\mathcal{F}_{\mathsf{OT}}$: Record $\mathcal{S}^*$'s inputs $\overline{\alpha}_1, \dots, \overline{\alpha}_h$, and send $\overline{K}_{\overline{\alpha}_i}^i$ for $i \in [h]$ to $\mathcal{S}^*$. Compute $\alpha := \sum_{i=0}^{h-1} 2^i \cdot \alpha_{h-i} \in [n]$.
6. Receive $\boldsymbol{\xi} \in \mathbb{F}_{2^{s'}}^n$ from $\mathcal{S}^*$ with $s' := \sigma + 2h$.
7. Compute $\Gamma := \langle \boldsymbol{\xi}, \mathbf{t} \rangle$ and send Γ to $\mathcal{S}^*$.
8. Define $\mathbf{u} \in \mathbb{Z}_{2^\ell}^n$ such that $u_\alpha = \beta$ and $u_i = 0$ for $i \in [n] \setminus \{\alpha\}$.
9. Sample $d \in_R \mathbb{Z}_{2^\ell}$ and send it to $\mathcal{S}^*$.
10. Define $\mathbf{w} \in \mathbb{Z}_{2^\ell}^n$ such that $w_i = v_i$ for $i \in [n] \setminus \{\alpha\}$ and $w_\alpha = \delta - d - \sum_{i\in[n]\setminus\{\alpha\}} w_i$.
11. $\mathcal{S}$ simulates the second call (Extend, 1) to $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ and receives $x, z \in \mathbb{Z}_{2^\ell}$ from $\mathcal{S}^*$.
12. Receive $\boldsymbol{\chi} \in \{0,1\}^n$ from $\mathcal{S}^*$. If $\mathrm{HW}(\boldsymbol{\chi}) \neq \frac{n}{2}$, abort.
13. Receive $x^* \in \mathbb{Z}_{2^\ell}$ from $\mathcal{S}^*$, and compute $x' := x + x^*$ (for an honest sender, we have $x' = \chi_\alpha \cdot \beta$).
14. Compute $V_{\mathcal{S}} := \sum_{i\in[n]} \chi_i \cdot w_i - z \in \mathbb{Z}_{2^\ell}$ (the honest $\mathcal{S}$'s input to $\mathcal{F}_{\mathsf{EQ}}$), and record $\mathcal{S}^*$'s actual input $V'_{\mathcal{S}} = V_{\mathcal{S}} + \varepsilon$ to $\mathcal{F}_{\mathsf{EQ}}$.
15. If $x' = \chi_\alpha \cdot \beta$:
    - If $V'_{\mathcal{S}} = V_{\mathcal{S}}$: Simulate successful equality check, and send $\mathbf{u}, \mathbf{w}$ as $\mathcal{S}$'s outputs to $\mathcal{F}_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$.
    - If $V'_{\mathcal{S}} \neq V_{\mathcal{S}}$: Simulate failing equality check and abort.
16. If $x' = \chi_\alpha \cdot \beta + \eta$ with $\eta \neq 0 \pmod{2^\ell}$: Let $v \in \mathbb{N}$ be maximal such that $2^v \mid \eta$. It must be $v < r$.
    - If $2^v \nmid \varepsilon$, then simulate a failing equality check and abort.
    - If $2^v \mid \varepsilon$, then compute

    $$\Delta' := \frac{\varepsilon}{2^v} \cdot \left(\frac{\eta}{2^v}\right)^{-1} \in \mathbb{Z}_{2^{r-v}}$$

    where the division is computed over $\mathbb{Z}$. If $r - v > s$ and $\Delta' \geq 2^s$, then simulate a failing equality check and abort. Otherwise set $s' := \min(s, r - v)$ and send (Guess, Δ', s') to $\mathcal{F}_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$. If it returns $\mathsf{success}$, then simulate a successful equality check. If it aborts, then simulate a failing equality check and abort.

17. If $\mathcal{S}^*$ sends (Guess, $\widetilde{\Delta}$) to the simulated $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, then $\mathcal{S}$ sends (Guess, $\widetilde{\Delta}$, s) to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ and returns the answer to $\mathcal{S}^*$. If $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ aborts, then $\mathcal{S}$ also aborts.

Claim 1 (Indistinguishability of the Simulation for Corrupted Sender)**.** *No PPT environment $\mathcal{Z}$ that chooses to corrupt the sender $\mathcal{S}$ can distinguish the real execution of the protocol from the simulation described above, except with probability* $\mathsf{negl}(\kappa)$.

Proof of Claim. The simulation of (Init) and (Extend), i.e., sending (Init) and (Extend) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, is identical to what would happen in the real protocol. Now we consider the simulation of ($\mathsf{SP\text{-}Extend}$, n):

The call (Extend, 1) to the simulated $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ results in $\mathcal{S}^*$ obtaining two values of its choice $a, c \in \mathbb{Z}_{2^\ell}$ – as in the real protocol. $\mathcal{S}$ creates the GGM tree honestly. Hence, the message $\overline{K}^{h+1}_1$ from the simulated receiver and the outputs that $\mathcal{S}^*$ receives from $\mathcal{F}_{\mathsf{OT}}$ are consistent and distributed as in the real protocol, and Γ is computed in the same way the real receiver would do it.

We leave the message d aside for a while, and focus on the remaining protocol first. From the second call (Extend, 1) to the simulated $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, $\mathcal{S}^*$ again obtains two values of its choice $x, z \in \mathbb{Z}_{2^\ell}$.

Finally, we must make sure that the simulated $\mathcal{F}_{\mathsf{EQ}}$ behaves as in the real protocol:

The honest $\mathcal{S}$ would input $V_{\mathcal{S}} := \sum_{i\in[n]} \chi_i \cdot w_i - z \in \mathbb{Z}_{2^\ell}$ to $\mathcal{F}_{\mathsf{EQ}}$, whereas the corrupted sender can send some arbitrary $V'_{\mathcal{S}} = V_{\mathcal{S}} + \varepsilon$. The $\mathcal{R}$ normally computes its input as $V_{\mathcal{R}} := \sum_{i\in[n]} \chi_i \cdot v_i - y \in \mathbb{Z}_{2^\ell}$, where y was computed as $y := y^* - \Delta \cdot x^*$, Δ and y^* were received from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and x^* from $\mathcal{S}^*$. So, in the simulation $\mathcal{S}$ does not know the right values of y^* and Δ.

$\mathcal{S}$ computes $x' := x + x^*$ (Step 13) and as noted above, we would have $x' = \chi_\alpha \cdot \beta$ if $\mathcal{S}^*$ behaved honestly. We obtain $\alpha \in [n]$ from the simulation of $\mathcal{F}_{\mathsf{OT}}$ (Step 5) and have received $\boldsymbol{\chi}$ from $\mathcal{S}^*$ (Step 12), so we know χ_α. $\mathcal{S}$ computes $\beta := a' + a$ from $\mathcal{S}^*$'s output a from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and its message a' to $\mathcal{R}$ (Step 2). While an honest $\mathcal{S}$ would compute $a' := \beta - a$, sending an arbitrary a' is just equivalent to choosing a different value for β. Hence, $\mathcal{S}$ knows $\chi_\alpha \cdot \beta$ and can check whether $x' = \chi_\alpha \cdot \beta$ ($\star$) holds.

If the equality ($\star$) holds (Step 15), $\mathcal{R}$ would input $V_{\mathcal{R}} = V_{\mathcal{S}}$ to $\mathcal{F}_{\mathsf{EQ}}$. Hence, we can simulate a successful equality check if $V'_{\mathcal{S}} = V_{\mathcal{S}}$, and a failing equality check with an abort otherwise – as in the real protocol.

On the other hand, if the equality ($\star$) does not hold (Step 16), define $\eta \neq 0 \mod 2^\ell$ such that $x' = \chi_\alpha \cdot \beta + \eta$. This means, $\mathcal{S}^*$ has send a corrupted x^*, but the equality check might nevertheless pass if the errors ε in $V'_{\mathcal{S}}$ and η in x' cancel out.

On one hand, we have $\mathcal{R}$'s input $V_{\mathcal{R}}$ to $\mathcal{F}_{\mathsf{EQ}}$ which depends on η:

$$\begin{aligned}V_{\mathcal{R}} &= \sum_{i\in[n]} \chi_i \cdot v_i - y \\ &= \sum_{i\in[n]} \chi_i \cdot v_i - y^* + \Delta \cdot x^* \\ &= \sum_{i\in[n]} \chi_i \cdot v_i - y^* + \Delta \cdot (x' - x) \\ &= \sum_{i\in[n]} \chi_i \cdot v_i - y^* + \Delta \cdot (\chi_\alpha \cdot \beta + \eta - x) \\ &= V_{\mathcal{S}} + \Delta \cdot \eta\end{aligned}$$

On the other hand, we have $\mathcal{S}^*$'s input $V'_{\mathcal{S}} = V_{\mathcal{S}} + \varepsilon$. So the equality test should pass if and only if

$$\Delta \cdot \eta = \varepsilon \pmod{2^\ell} \tag{2.1}$$

holds. While $\mathcal{S}$ does not know the right value of Δ, it can use the global key query of $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$.

As above, define $v \in \mathbb{N}$ such that 2^v is the largest power of two that divides η. For Equation (2.1) to hold, 2^v must also divide ε. If this is not the case, $\mathcal{S}$ can safely simulate a failing equality test. Assuming 2^v divides both η and ε, we can divide both sides of Equation (2.1) by 2^v and reduce the modulus accordingly.

$$\Delta \cdot \frac{\eta}{2^v} = \frac{\varepsilon}{2^v} \pmod{2^{r-v}}$$

Since $\frac{\eta}{2^v}$ must be odd, we can solve for Δ.

$$\Delta = \frac{\varepsilon}{2^v} \cdot \left(\frac{\eta}{2^v}\right)^{-1} \pmod{2^{r-v}}$$

Let $\Delta' := \frac{\varepsilon}{2^v} \cdot \left(\frac{\eta}{2^v}\right)^{-1} \bmod 2^{r-v}$ be the result of the computation. We know that $\Delta \in \mathbb{Z}_{2^s}$. So, if $r - v \geq s$, then it must be $\Delta' = \Delta$ for the equality check to hold, and the abort in case $\Delta' \geq 2^s$ is safe. If $r - v < s$, then Δ' must consist of the lower $r - v$ bits of Δ for the check to hold. Hence, the global key query $(\mathsf{Guess}, \Delta', \min(s, r - v))$ of $\mathcal{S}$ results in an abort of $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ exactly when the equality test would fail.

To summarize, all messages (and abort) are distributed perfectly indistinguishable in protocol and simulation, except for d. We now show that any distinguisher $\mathcal{Z}$ for malicious senders can break the pseudorandomness (Definition 3) of the PPRF. For this, consider that in the real protocol $\mathcal{R}$ computes $d := \gamma - \sum_{j=1}^{n} v_j \in \mathbb{Z}_{2^\ell}$, where $\gamma := b - \Delta \cdot a'$ and $\mathcal{R}$ received Δ, b from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and a' from $\mathcal{S}$. On the other hand, in the simulation it is sampled uniformly at random as $d \in_R \mathbb{Z}_{2^\ell}$. The reason is that in the simulation, $\mathcal{S}$ never learns the secret key Δ that the ideal $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ has generated for the honest $\mathcal{R}$ – only the environment learns it.

We can now use any PPT environment $\mathcal{Z}$ that can distinguish both to break pseudorandomness as follows: First, observe that $\mathcal{Z}$ is fixed and that a reduction can make manipulations to the inner state of its security game. We construct a special simulator $\widehat{\mathcal{S}}$. Consider an execution of $\mathcal{Z}$ using $\widehat{\mathcal{S}}$ and let it first extract the point α where $\mathcal{S}$ will puncture the PPRF in the protocol from the hybrid $\mathcal{F}_{\mathsf{OT}}$.

$\widehat{\mathcal{S}}$ forwards α to the PPRF security game as in Definition 3 and obtains all seeds $\overline{K}^i$ for keys that it will input into the $\mathcal{F}_{\mathsf{OT}}$ to deliver to the sender.

It also obtains a point v^*_α which by the security game is either $F(k, \alpha)$ or uniformly random. Since our reduction can control the security experiment of $\mathcal{Z}$ and therefore controls $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$ it has access to Δ that is generated by the functionality. It computes d as $d = \gamma - v^*_\alpha - \sum_{j=1,j\neq\alpha}^{n} v_j$ and uses d as its message to $\mathcal{S}$. For all other purposes, $\widehat{\mathcal{S}}$ just acts like $\mathcal{S}$.

By construction, if v^*_α is uniformly random then d is distributed as in the simulation, while if $v^*_\alpha = F(k, \alpha)$ then d is identical to the real protocol. Hence any $\mathcal{Z}$ that distinguishes both breaks Definition 3. By assumption, any PPT algorithm (in κ) can only do so with probability at most $\mathsf{negl}(\kappa)$. ■

Malicious Receiver. The simulation is setup as follows: $\mathcal{S}$ simulates a party $\mathcal{R}^*$ in its head and gives control to $\mathcal{Z}$, and sends $(\mathsf{corrupt}, \mathcal{R})$ to $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$. It also simulates instances of $\mathcal{F}^{\ell,s}_{\text{vole2k}}$, $\mathcal{F}_{\mathsf{OT}}$, and $\mathcal{F}_{\mathsf{EQ}}$.

For the call to (Init), $\mathcal{S}$ receives $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{R}^*$, and sends Δ to $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$. Calls to (Extend) are simply forwarded to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$, and $\mathcal{S}$ can just simulate the interaction with the ideal functionality. The main part of proof is the simulation of $(\mathsf{SP\text{-}Extend}, n)$. Let $h := \lceil \log n \rceil$.

1. $\mathcal{S}$ simulates the call $(\mathsf{Extend}, 1)$ to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$ and receives $b \in \mathbb{Z}_{2^\ell}$ from $\mathcal{R}^*$.
2. Send $a' \in_R \mathbb{Z}_{2^\ell}$ to $\mathcal{R}^*$.
3. Receive $\overline{K}^{h+1}_1$ from $\mathcal{R}^*$.
4. Simulates the calls to $\mathcal{F}_{\mathsf{OT}}$ and records $\mathcal{R}^*$'s inputs $(\overline{K}^i_0, \overline{K}^i_1)_{i\in[h]}$.
5. Sample $\boldsymbol{\xi} \in_R \mathbb{F}^n_{2^{s'}}$ with $s' := \sigma + 2h$, and send $\boldsymbol{\xi}$ to $\mathcal{R}^*$.
6. Receive Γ from $\mathcal{R}^*$.
7. For $\alpha \in [n]$, compute $\mathbf{v}^\alpha \leftarrow \mathsf{GGM.Eval}(n, \alpha, (\overline{K}^1_{\overline{\alpha}_1}, \dots, \overline{K}^h_{\overline{\alpha}_h}, \overline{K}^{h+1}))$.
8. Define

$$I := \{\alpha \in [n] \mid \mathsf{GGM.Check}(n, \alpha, (\overline{K}^1_{\overline{\alpha}_1}, \dots, \overline{K}^h_{\overline{\alpha}_h}, \overline{K}^{h+1}), \boldsymbol{\xi}, \Gamma) = \top\}.$$

$\mathcal{S}$ sends I to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$. If it aborts, then simulate the abort of $\mathcal{S}$.

9. If the outputs $\mathbf{v}^\alpha$ are not consistent, i.e., $v_j^\alpha \neq v_j^{\alpha'}$ for some $\alpha \neq \alpha' \in I$ and $j \in [n] \setminus \{\alpha, \alpha'\}$, then abort the simulation (by Theorem 2, this should happen with negligible probability).

10. Case $|I| = 1$: $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ sends $(\mathsf{success}, \beta)$. With this information, we can basically follow the remaining protocol:

    (a) Note that we now know the real α that $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ has chosen since $I = \{\alpha\}$. Hence, we can compute $\mathbf{u} \in \mathbb{Z}^n_{2^\ell}$ such that $u_\alpha = \beta$ and $u_i = 0$ for all $i \neq \alpha$.

    (b) Moreover, compute $a := \beta - a'$, and $\delta := \Delta \cdot a + b$.

    (c) Receive $d' \in \mathbb{Z}_{2^\ell}$ from $\mathcal{R}^*$.

    (d) Compute $\mathbf{w} \in \mathbb{Z}^n_{2^\ell}$ such that $w_i := v_i$ for all $i \neq \alpha$, and $w_\alpha := \delta - d' - \sum_{i \neq \alpha} w_i$.

    (e) $\mathcal{S}$ simulates the second call $(\mathsf{Extend}, 1)$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and receives $y^* \in \mathbb{Z}_{2^\ell}$ from $\mathcal{R}^*$.

    (f) Sample $x \in_R \mathbb{Z}_{2^\ell}$ and set $z := \Delta \cdot x + y^*$.

    (g) Sample $\boldsymbol{\chi} \in_R \{0,1\}^n$ with $\mathrm{HW}(\boldsymbol{\chi}) = \frac{n}{2}$ and compute $x^* := \chi_\alpha \cdot \beta - x$ and send them to $\mathcal{R}^*$.

    (h) Compute $V_\mathcal{S} := \sum_{i=1}^n \chi_i \cdot w_i - z$. Record the value $V'_\mathcal{R}$ that $\mathcal{R}^*$ sends to $\mathcal{F}_{\mathsf{EQ}}$.

    (i) If $V_\mathcal{S} \neq V'_\mathcal{R}$: Make $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ abort by sending $(\mathsf{query}, \emptyset)$, and simulate failing equality test with $V_\mathcal{S}$ as $\mathcal{S}$'s input.

    (j) If $V_\mathcal{S} = V'_\mathcal{R}$: Send $(\mathsf{continue})$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and simulate successful equality test.

    (k) Compute $\mathbf{v} := \mathbf{w} - \Delta \cdot \mathbf{u}$, and send $\mathbf{v}$ to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ as $\mathcal{R}$'s output.

11. Case $|I| > 1$: $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ sends $(\mathsf{success})$

    (a) Let $v_1, \ldots, v_n$ denote the leaves of the GGM tree. (We can recover these from $(v_j^\alpha)_{j \in [n] \setminus \{\alpha\}}$ and $(v_j^{\alpha'})_{j \in [n] \setminus \{\alpha'\}}$ for two different $\alpha, \alpha' \in I$.

    (b) Define $d := (b - \Delta \cdot a') - \sum_{j=1}^n v_j \in \mathbb{Z}_{2^\ell}$, i.e., the value that an honest $\mathcal{R}$ would send. Receive $d' = d + \varepsilon \in \mathbb{Z}_{2^\ell}$ from $\mathcal{R}^*$, where ε denotes a possible error added by $\mathcal{R}^*$.

    (c) $\mathcal{S}$ simulates the second call $(\mathsf{Extend}, 1)$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and receives $y^* \in \mathbb{Z}_{2^\ell}$ from $\mathcal{R}^*$.

    (d) Sample $\boldsymbol{\chi} \in_R \{0,1\}^n$ with $\mathrm{HW}(\boldsymbol{\chi}) = \frac{n}{2}$ and $x^* \in_R \mathbb{Z}_{2^\ell}$ and send them to $\mathcal{R}^*$.

    (e) Compute $V_\mathcal{R} := \sum_{i=1}^n \chi_i \cdot v_i - (y^* - \Delta \cdot x^*) \in \mathbb{Z}_{2^\ell}$, i.e., the value an honest $\mathcal{R}$ would send to $\mathcal{F}_{\mathsf{EQ}}$. Record the actual value $V'_\mathcal{R}$ that $\mathcal{R}^*$ sends to $\mathcal{F}_{\mathsf{EQ}}$.

    (f) If $\varepsilon = 0$:

        - If $V'_\mathcal{R} = V_\mathcal{R}$: Send $(\mathsf{continue})$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and simulate successful equality test.
        - If $V'_\mathcal{R} \neq V_\mathcal{R}$: Send $(\mathsf{query}, \emptyset)$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and simulate failing equality test with $V_\mathcal{R}$ as $\mathcal{S}$'s input.

(g) If $\varepsilon \neq 0$:
- If $V'_{\mathcal{R}} = V_{\mathcal{R}}$, define $J := \{i \in [n] \mid \chi_i = 0\}$ and $V^* := V_{\mathcal{R}} - \varepsilon$.
- If $V'_{\mathcal{R}} = V_{\mathcal{R}} - \varepsilon$, define $J := \{i \in [n] \mid \chi_i = 1\}$ and $V^* := V_{\mathcal{R}}$.
- If $V'_{\mathcal{R}} \notin \{V_{\mathcal{R}}, V_{\mathcal{R}} - \varepsilon\}$, define $J := \emptyset$. Sample $b \in_R \{0,1\}$ and set $V^* := V_{\mathcal{R}} - b \cdot \varepsilon$. Although the equality check will never pass, we cannot abort directly, since $\mathcal{R}^*$ learns $\mathcal{S}$'s input. Hence, we need to supply a value that looks right, where the error ε is subtracted with 1/2 probability corresponding to the probability that $\chi_\alpha = 1$.

Send (query, J) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. If it aborts, simulate failing equality test with V^* as $\mathcal{S}$'s input and simulate $\mathcal{S}$ aborting. Otherwise, receive $\alpha \in [n]$ from $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ and simulate successful equality test.

(h) We now know the correct value α chosen by the ideal functionality. Let $\mathbf{v}' \in \mathbb{Z}^n_{2^\ell}$ such that $v'_\alpha = v_\alpha - \varepsilon$ and $v'_i = v_i$ for $i \in [n] \setminus \{\alpha\}$. $\mathcal{S}$ sends $\mathbf{v}'$ as $\mathcal{R}$'s output to $\mathcal{F}^{2,s}_{\mathsf{vole2k}}$.

Claim 2 (Indistinguishability of the Simulation for Corrupted Receiver)**.** *No PPT environment $\mathcal{Z}$ that chooses to corrupt the receiver $\mathcal{R}$ can distinguish the real execution of the protocol from the simulation described above except with probability* $2^{-(\sigma+1)}$.

Proof of Claim. The simulation of Init, i.e., sending (Init) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, results again in a view of $\mathcal{R}^*$ which is identical to that in the real protocol. Note that $\mathcal{S}$ now learns the global key Δ since it is chosen by $\mathcal{R}^*$.

Now we consider the simulation of (Extend, n) with $n = 2^h$.

The call $(\mathsf{Extend}, 1)$ to the simulated $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ results in $\mathcal{R}^*$ obtaining a value of its choice $b \in \mathbb{Z}_{2^\ell}$ – as in the real protocol.

The first difference already occurs in Step 2, where the simulator sends a randomly samples $a' \in_R \mathbb{Z}_{2^\ell}$, whereas the real sender $\mathcal{S}$ would compute $a' := \beta - a$. In the simulation we don't know β, but a is distributed uniformly at random. Hence, from V^*'s view a' is distributed identically in both cases.

If we learn the correct β from $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ later on (we will, e.g., in Step 10), then we can compute $a := \beta - a'$ and $c := \Delta \cdot a + b$ and pretend that we received (a, c) from $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ during the first call to $(\mathsf{Extend}, 1)$, and these values will still be consistent with the view of $\mathcal{R}^*$.

The next Steps 3-6 simulate the transfer of the punctured PRF key. The message $\boldsymbol{\xi}$ send in Step 5 is sampled in the same way as the real $\mathcal{R}$ would do it.

Now we need to make sure that $\mathcal{S}$ simulates an abort if and only iff the consistency check of the GGM tree fails. In the simulation, we don't know the real value of α that the ideal functionality chooses, but we can collect all possible values of α for which the check passes in the set I. In the real execution, $\mathcal{S}$ would abort if and only if the real α is not contained in this set. Hence, in Step 8 we use the first query and send I to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$.

In the case that $\mathcal{R}^*$ has managed to create an inconsistent GGM tree which passes the check, then $\mathcal{S}$ aborts the simulation in Step 9, but by Theorem 2, this happens only

with probability $2^{-(\sigma+1)}$. Therefore, we assume in the following that the GGM tree is consistent.

In case $|I| = 1$ (Step 10), we have recovered the correct value of α since it must be $I = \{\alpha\}$, and $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ sends use the correct value of β. So we can compute $\mathcal{S}$'s output $\mathbf{u}$ (Step 10a) and values a and δ which are consistent with the simulation (Step 10b). Then $\mathcal{S}$ can behaves in the same way as the real $\mathcal{S}$, and simulate the equality test accordingly. Since $\mathcal{S}$ can compute $\mathbf{w}$ in Step 10d and already knows Δ and $\mathbf{u}$, it can successfully recover $\mathbf{v}$ (Step 10k) and send it to $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ to select $\mathcal{R}$'s output.

In case $|I| > 1$ (Step 11), we do not learn α and β from the query. We can, however, compute $\mathbf{v}$ in Step 11a, since we have (at least) two different $\mathbf{v}^{\alpha}$, $\mathbf{v}^{\alpha'}$. This allows us to compute the value d that an honest $\mathcal{R}$ would send, when we receive the (possibly maliciously chosen) message $d' = d + \varepsilon$ from $\mathcal{R}^*$ (Step 11b). If $\varepsilon \neq 0$, then we know that $\mathcal{R}^*$ is cheating, but we can only abort the simulation if $\mathcal{R}^*$ is also caught by the check in the real protocol.

From the simulation of the $(\textsf{Extend}, 1)$ call to $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$, we learn the value y^* that $\mathcal{R}^*$ has chosen. The coefficient vector $\boldsymbol{\chi}$ is sampled uniformly at random as in the real protocol.

Since $\mathcal{S}$ uses the received d to compute the value of w at position α, the error ε is propagated and $\mathcal{S}$ instead computes $w'_\alpha = w_\alpha - \varepsilon$, and $\mathcal{S}$'s input to $\mathcal{F}_{\textsf{EQ}}$ is $V'_{\mathcal{S}} = V_{\mathcal{S}} - \chi_\alpha \cdot \varepsilon$. Hence, to make the equality check pass, $\mathcal{R}^*$ needs to adjust its input value $V'_{\mathcal{R}}$ to account for the error ε iff $\chi_\alpha = 1$. Otherwise, it need to send the same value $V_{\mathcal{R}}$ that the honest $\mathcal{R}$ would send. We can compute this value (Step 11e).

If $\varepsilon = 0$ ($\mathcal{R}^*$ has sent the correctly computed d, covered in Step 11f), we know that the sender's input $V_{\mathcal{S}}$ matches the honest $\mathcal{R}$'s input $V_{\mathcal{R}}$. Hence, we can simulate a successful equality test if $\mathcal{R}^*$'s actual input $V'_{\mathcal{R}} = V_{\mathcal{R}}$, and simulate an abort otherwise.

If $\varepsilon \neq 0$ ($\mathcal{R}^*$ has sent an incorrect value $d' = d + \varepsilon$, covered in Step 11g), then we know that it must be $V_{\mathcal{S}} = V_{\mathcal{R}} - \chi_\alpha \cdot \varepsilon$ for the check to pass. So, if $V'_{\mathcal{R}} \notin \{V_{\mathcal{R}}, V_{\mathcal{R}} - \varepsilon\}$, we can simulate a failing equality test, where the sender used either of the values as input with probability $1/2$. Since we do not know α, we cannot just lookup the value of χ_α in $\boldsymbol{\chi}$, but we can make a query to the ideal $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ whether $\alpha \in J$ for some index set J. If $V'_{\mathcal{R}} = V_{\mathcal{R}}$, we set J as the set of indices where $\boldsymbol{\chi}$ is 0, and if $V'_{\mathcal{R}} = V_{\mathcal{R}} - \varepsilon$, we set J as the set of indices where $\boldsymbol{\chi}$ is 1. Hence, J will contain exactly those values for α for which the the equality check would pass: if $V'_{\mathcal{R}} = V_{\mathcal{R}}$, then it must be $\chi_\alpha = 0$ so that the error ε disappears, and if $V'_{\mathcal{R}} = V_{\mathcal{R}} - \varepsilon$, then it must be $\chi_\alpha = 1$ so that ε is propagated. Hence, the equality test passes iff $\alpha \in J$. So we query the ideal $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ and simulate a failing test if it aborts. Otherwise, it sends us the value of α, and we can adjust the position α in $\mathcal{R}^*$'s output $\mathbf{v}$ corresponding to the error ε (Step 11h).

The simulation is perfect unless $\mathcal{S}$ aborts it, which happens with probability at most $2^{-(\sigma+1)}$ if $\mathcal{R}^*$ manages to create an inconsistent GGM tree which passes the check. ■

Theorem 3 follows from the combination of the two claims stating the indistinguishability of the simulations. □

2.3.3 Protocol Complexity

Let $n \in \mathbb{N}$, $h := \lceil \log n \rceil$, and $\sigma' := \sigma + 2h$. For one call to $(\mathsf{SP\text{-}Extend}, n)$, we

- use $2\times$ VOLE (of length $1 \times \mathbb{Z}_{2^\ell}$),
- use $h\times$ OT (on strings of length κ),
- use $1\times$ EQ for $\mathbb{Z}_{2^\ell}$ elements,
- transfer $3\times$ $\mathbb{Z}_{2^\ell}$ elements,
- transfer $2\times$ $\mathbb{F}_{2^{\sigma'}}$ elements, and
- transfer $2\times$ $\{0,1\}^\kappa$ PRG seeds.

Overall, using the equality test sketched in Section 2.2.3 and Silent OT [BCG+19a, YWL+20, CRR21], we transfer $4\ell + 2\sigma + 4\lceil \log n \rceil + (5 + 2\lceil \log n \rceil)\kappa$ bit plus the costs of 2 VOLEs.

2.4 Vector OLE Construction

Given our single-point VOLE protocol, we build a protocol for random VOLE extension over $\mathbb{Z}_{2^\ell}$ by running t single-point instances of length n/t, and concatenating their outputs to obtain a weight t VOLE correlation of length n. Then, these (together with some additional VOLEs) can be extended into pseudorandom VOLEs by applying the primal LPN assumption over $\mathbb{Z}_{2^\ell}$ with regular noise vectors of weight t. Since our single-point protocol introduces some leakage on the hidden point, we need to rely on a variant of LPN with some leakage on the regular noise coordinates.

2.4.1 Leaky Regular LPN Assumption

The assumption, below, translates the leakage from the single-point VOLE functionality (Figure 2.4) into leakage on the LPN error vector. Note that there are two separate leakage queries: the first of these allows the adversary to try and guess a single predicate on the entire noise vector, and aborts if this guess is incorrect. This is similar to previous works [BCG+19a, WYKW21a], and essentially only leaks 1 bit of information on average on the position of the non-zero entries. The second query, in Step 5 is more powerful, since for each query made by the adversary, the exact position of one noise coordinate is leaked with probability 1/2. Intuitively, this means that up to c coordinates of the error vector can be leaked with probability 2^{-c}.

Definition 6. *Let $\mathbf{A} \leftarrow \mathsf{G}(m, n, 2^\ell) \in \mathbb{Z}_{2^\ell}^{m \times n}$ be a primal-LPN matrix, and consider the following game $G_b(\kappa)$ with a PPT adversary $\mathcal{A}$, parameterized by a bit b and security parameter κ:*

1. *Sample $\mathbf{e} = (\mathbf{e}_1, \ldots, \mathbf{e}_t) \leftarrow \mathbb{Z}_{2^\ell}^n$, where each sub-vector $\mathbf{e}_i \in \mathbb{Z}_{2^\ell}^{n/t}$ has exactly one non-zero entry in $\mathbb{Z}_{2^\ell}^*$, in position α_i, and sample $\mathbf{s} \leftarrow \mathbb{Z}_{2^\ell}^m$ uniformly*

2. *$\mathcal{A}$ sends sets $I_1, \dots, I_t \subset [n/t]$*

3. *If $\alpha_j \in I_j$ for all $j \in [t]$, send OK to $\mathcal{A}$, otherwise abort. Additionally, for any j where $|I_j| = 1$, send $\mathbf{e}_j$ to $\mathcal{A}$*

4. *$\mathcal{A}$ sends sets $J_1, \dots, J_t \subset [n/t]$*

5. *For each J_i where $|J_i| = n/(2t)$: if $\alpha_i \in J_i$, send α_i to $\mathcal{A}$, otherwise abort*

6. *Let $\mathbf{y}_0 = \mathbf{s} \cdot \mathbf{A} + \mathbf{e}$ and sample $\mathbf{y}_1 \leftarrow \mathbb{Z}_{2^\ell}^n$*

7. *Send $\mathbf{y}_b$ to $\mathcal{A}$*

8. *$\mathcal{A}$ outputs a bit b' (if the game aborted, set the output to $\perp$)*

The assumption is that $|\Pr[\mathcal{A}^{G_0(\kappa)} = 1] - \Pr[\mathcal{A}^{G_1(\kappa)} = 1]|$ is negligible in κ.

We now discuss how the additional leakage in our leaky variant of the LPN assumption (Definition 6) affects the hardness of the problem.

The Security Game.

Recall the security game from Definition 6 where the adversary $\mathcal{A}$ can make two queries for the indices of the non-zero entries in the error vector $\mathbf{e}$, an I-query (Step 2) and a J-query (Step 4).

For each query, $\mathcal{A}$ sends a collection of sets $I_1, \dots, I_t \subseteq [\frac{n}{t}]$ (resp. $J_{i_1}, \dots, J_{i_h} \subseteq [\frac{n}{t}]$ with $h \in [0, t]$ and all $i_j \in [t]$ different) to the challenger. The adversary then learns whether $\alpha_i \in I_i$ for all $i \in [t]$ (resp. $\alpha_{i_j} \in J_{i_j}$ for all $j \in [h]$) where α_i is the index of the non-zero entry in the subvector $\mathbf{e}_i$. If this is not the case, i.e., there is an index $i \in [t]$ such that $\alpha_i \notin I_i$ (resp. i_j such that $\alpha_{i_j} \notin J_i$), then the game aborts.

The I-query is similar to previous works [BCG+19a, WYKW21a], and essentially only leaks 1 bit of information on average on the position of the non-zero entries: The adversary then learns whether $\alpha_i \in I_i$ for all $i \in [t]$. In the case that one of the sets contain only the correct index $I_i = \{\alpha_i\}$, our variant additionally reveals the non-zero value β_i. Compared to previous works, the adversary is then able to remove the noise and, thus, learns $\frac{n}{t}$ instead of $\frac{n}{t} - 1$ noiseless equations.

The J-query is more powerful: For each $i \in [t]$, the adversary has the option to make a guess by sending a subset $J_i \subseteq [\frac{n}{t}]$ of size $\frac{n}{2t}$ (sets of other sizes are ignored). In contrast to the I-query, the adversary learns the correct index α_i for each successful guess J_i of this form. If there is a guess J_i of this form such that $\alpha_i \notin J_i$, the game aborts. However, due to the size restriction, every guess independently succeeds with probability $1/2$. Hence, except with probability at most $2^{-\sigma}$, the adversary will not learn more than σ noisy coordinates α_i without without the game aborting.

Estimating Security of Leaky LPN.

Let $h(i, j) := (i-1) \cdot (\frac{n}{t}) + j$ be an index function which computes the index of a length n vector that corresponds to the jth entry of the ith subvector of length $\frac{n}{t}$. We use $\mathbf{A}_k$ to denote the kth column of $\mathbf{A}$.

Suppose the adversary has received an LPN sample $\mathbf{y} = \mathbf{s} \cdot \mathbf{A} + \mathbf{e}$ and learned the position α_i of the noise in block $\mathbf{e}_i$. Then it knows that $y_{h(i,\alpha_i)} = \mathbf{s} \cdot \mathbf{A}_{h(i,\alpha_i)} + \beta_i$ holds for the (unknown) noise value β_i. It also knows that $y_{h(i,j)} = \mathbf{s} \cdot \mathbf{A}_{h(i,j)}$ holds for all other indices in this block $j \in [\frac{n}{t}] \setminus \{\alpha_i\}$, i.e., it now has $\frac{n}{t} - 1$ linear equations of the secret vector $\mathbf{s}$ *without noise*. In the worst case, these could be used to recover $\frac{n}{t} - 1$ entries of $\mathbf{s}$.

Now the adversary can to learn up to σ noisy coordinate except with negligible probability. Therefore, we must tolerate the leakage of up to $\sigma \cdot (\frac{n}{t} - 1)$ noise-free equations or the same number of entries of $\mathbf{s}$. Hence, it can transform the given LPN instance into a smaller instance where the σ affected blocks are removed and the secret is $\sigma \cdot (\frac{n}{t} - 1)$ entries smaller. and we require that it is still infeasible for the adversary to solve this smaller LPN instance.

To summarize, we assume that an instance of the leaky LPN problem with parameters (m, t, n) is as hard as a standard LPN instance (Definition 1) with reduced parameters $(m', t', n') = (m - \sigma \cdot (\frac{n}{t} - 1), t - \sigma, n - \sigma \cdot \frac{n}{t})$. Hence, we must choose (m, t, n) such that n is large enough for our application and that the (standard) regular LPN problem with parameters (m', t', n') is hard to solve.

Estimating Security of Standard LPN.

We initially estimated the security of standard regular LPN (Definition 1) with the reduced parameters (m', t', n') following Boyle et al. [BCGI18] as

$$\log_2 \left(\frac{m' + 1}{(1 - \frac{m'}{n'})^{t'}} \right) \text{ bits,}$$

based on their estimation of the cost of the low-weight parity-check attack.

Recently, Liu et al. [LWYY22] found that the above estimate is very conservative, and the pooled Gauss attack performs better for all practical parameters. They provide a script to compute more precise estimates, and we refer to their work for more details. As mentioned in Section 2.2.5, by choosing the LPN noise values to be odd, we avoid the reduction attack from Liu et al., which would otherwise halve the noise rate.

2.4.2 Vector OLE Protocol

Our complete VOLE protocol is given in Figure 2.6. It realises the functionality $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ (Figure 2.1), which is the same functionality used for base VOLEs in our single-point protocol. This allows us to use the same kind of "bootstrapping" mechanism as [WYKW21a], where a portion of the produced VOLE outputs is reserved to be used as the base VOLEs in the next iteration of the protocol.

In the Init phase of the protocol, the parties create a base VOLE of length m, defining the random LPN secret $\mathbf{u}$, given to the sender, and the scalar Δ, given to the receiver. Then, in each call to Extend, the parties run t instances of $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$ to generate $\mathbf{c} = (c_1, \ldots, c_t)$ and $\mathbf{e} = (e_1, \ldots, e_t)$ for the sender and $\mathbf{b} = (b_1, \ldots, b_t)$ for the receiver. The sender then simply computes $\mathbf{x} \leftarrow \mathbf{u} \cdot \mathbf{A} + \mathbf{e} \in \mathbb{Z}^n_{2^r}$ and $\mathbf{z} \leftarrow \mathbf{w} \cdot \mathbf{A} + \mathbf{c} \in \mathbb{Z}^n_{2^r}$ and the receiver computes $\mathbf{y} = \mathbf{v} \cdot \mathbf{A} + \mathbf{b} \in \mathbb{Z}^n_{2^r}$. This results in the sender holding $\mathbf{x}, \mathbf{z}$ and the receiver holding $\mathbf{y}$ such that $\mathbf{z} = \mathbf{x} \cdot \Delta + \mathbf{y}$. The first m entries of these are reserved to define a fresh LPN secret for the next call to Extend, while the remainder are output by the parties.[5]

Theorem 4. *The protocol $\Pi^{\ell,s}_{\mathsf{vole2k}}$ in Fig. 2.6 securely realizes the functionality $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$ in the $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$-hybrid model, under the leaky regular LPN assumption.*

Proof of Theorem 4. We first consider the case of a corrupt sender, and then separately a corrupt receiver.

Malicious sender. We construct a simulator as follows. In the Init phase, the simulator first forwards (Init) to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$, and then receives $\mathbf{u}, \mathbf{w}$ from $\mathcal{A}$, as its input to the (Extend) command of $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$.

In the Extend phase, the simulator receives vectors $\mathbf{e}_i, \mathbf{c}_i$ from $\mathcal{A}$, for $i \in [t]$, and defines $\mathbf{e} = (\mathbf{e}_1, \ldots, \mathbf{e}_t)$, $\mathbf{c} = (\mathbf{c}_1, \ldots, \mathbf{c}_t)$. The simulator then computes $\mathbf{x}, \mathbf{z}$ as in the protocol, updates the vectors $\mathbf{u}, \mathbf{w}$ accordingly, and finally sends $(\mathbf{x}[m:n), \mathbf{z}[m:n)) \in \mathbb{Z}^\ell_{2^\ell} \times \mathbb{Z}^\ell_{2^\ell}$ as input to the $\mathcal{F}^{\ell,s}_{\text{vole2k}}$ functionality.

Whenever $\mathcal{A}$ sends a key query command to $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$, the simulator sends the query to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$ and forwards its response to $\mathcal{A}$. If $\mathcal{F}^{\ell,s}_{\text{vole2k}}$ aborts, the simulator aborts.

Indistinguishability. Since there is no interaction in the protocol, and in the ideal world, the outputs of both parties are computed the exact same way as the real protocol, it is clear that the two executions are perfectly indistinguishable.

Malicious receiver. To simulate the Init phase, the simulator first receives Δ from $\mathcal{A}$ and forwards this to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$, and then receives $\mathbf{v} \in \mathbb{Z}^m_{2^\ell}$ from $\mathcal{A}$.

In the Extend phase, the simulator samples a random noise vector $(\mathbf{e}_1, \ldots, \mathbf{e}_t)$, and uses this to respond to the leakage queries from $\mathcal{A}$, just as $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$ would. If any query aborts, it sends abort to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$ and aborts. If the queries are successful, it receives $\mathbf{b}_i$ from $\mathcal{A}$, for $i \in [t]$, then defines $\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_t)$. It then computes $\mathbf{y} = \mathbf{v} \cdot \mathbf{A} + \mathbf{b}$, updates $\mathbf{v}$ as an honest $\mathcal{R}$ would, and sends the last $n - m$ entries of $\mathbf{y}$ to $\mathcal{F}^{\ell,s}_{\text{vole2k}}$.

Indistinguishability. First, note that the probability of abort is identical in both the real and ideal executions, since the simulator responds to the leakage queries using a

[5] In our implementation, we actually reserve $m + 2t$ of the outputs, since we need 2 extra VOLEs for each execution of the protocol for $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$.

VOLE for $\mathbb{Z}_{2^k}$: $\Pi^{\ell,s}_{\mathsf{vole2k}}$

Parameters Fix some parameters:

- n: LPN output size
- m: LPN secret size
- t: number of error coordinates for LPN (assume that $t \mid n$)
- n/t: size of a block in regular LPN
- $\mathbf{A} \in \mathbb{Z}_{2^\ell}^{m \times n}$ is the generator matrix used in primal-LPN

Init This must be called by the parties first and is executed once.

1. $\mathcal{S}$ and $\mathcal{R}$ send (Init) to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$, and $\mathcal{R}$ receives $\Delta \in \mathbb{Z}_{2^s}$.
2. $\mathcal{S}$ and $\mathcal{R}$ send (Extend, m) to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$. $\mathcal{S}$ receives $\mathbf{u}, \mathbf{w} \in \mathbb{Z}_{2^\ell}^m$, and $\mathcal{R}$ receives $\mathbf{v} \in \mathbb{Z}_{2^\ell}^m$, such that $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$ over $\mathbb{Z}_{2^\ell}$.

Extend This protocol can be executed multiple times.

1. For $i \in [t]$, $\mathcal{S}$ and $\mathcal{R}$ send $(\mathsf{SP\text{-}Extend}, n/t)$ to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ which returns $\mathbf{e}_i, \mathbf{c}_i$ to $\mathcal{S}$ and $\mathbf{b}_i$ to $\mathcal{R}$ such that $\mathbf{c}_i = \Delta \cdot \mathbf{e}_i + \mathbf{b}_i$ over $\mathbb{Z}_{2^\ell}^{n/t}$, and $\mathbf{e}_i \in \mathbb{Z}_{2^\ell}^{n/t}$ has exactly one entry invertible modulo 2^ℓ and zeros everywhere else.
2. Define $\mathbf{e} := (\mathbf{e}_1, \dots, \mathbf{e}_t) \in \mathbb{Z}_{2^\ell}^n$, $\mathbf{c} := (\mathbf{c}_1, \dots, \mathbf{c}_t) \in \mathbb{Z}_{2^\ell}^n$, and $\mathbf{b} := (\mathbf{b}_1, \dots, \mathbf{b}_t) \in \mathbb{Z}_{2^\ell}^n$. Then $\mathcal{S}$ computes $\mathbf{x} := \mathbf{u} \cdot \mathbf{A} + \mathbf{e} \in \mathbb{Z}_{2^\ell}^n$, and $\mathbf{z} := \mathbf{w} \cdot \mathbf{A} + \mathbf{c} \in \mathbb{Z}_{2^\ell}^n$. $\mathcal{R}$ computes $\mathbf{y} := \mathbf{v} \cdot \mathbf{A} + \mathbf{b} \in \mathbb{Z}_{2^\ell}^n$.
3. $\mathcal{S}$ updates $\mathbf{u}, \mathbf{w}$ by setting $\mathbf{u} := \mathbf{x}[0 : m) \in \mathbb{Z}_{2^\ell}^m$ and $\mathbf{w} := \mathbf{z}[0 : m) \in \mathbb{Z}_{2^\ell}^m$, and outputs $(\mathbf{x}[m : n), \mathbf{z}[m : n)) \in \mathbb{Z}_{2^\ell}^\ell \times \mathbb{Z}_{2^\ell}^\ell$. $\mathcal{R}$ updates $\mathbf{v}$ by setting $\mathbf{v} := \mathbf{y}[0 : m) \in \mathbb{Z}_{2^\ell}^m$ and outputs $\mathbf{y}[m : n) \in \mathbb{Z}_{2^\ell}^\ell$.

Figure 2.6: Protocol for VOLE over $\mathbb{Z}_{2^k}$ in the $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$-hybrid model. Based on [WYKW21a].

random noise vector, just as $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$ does in the real execution. It remains to show that the distribution of the parties' outputs in the ideal execution is indistinguishable from the actual protocol.

For simplicity, we consider the case of a single call to Extend; handling multiple calls follows with a standard hybrid argument, and the fact that the LPN secret used in subsequent calls is independent of previous outputs. Suppose there is an environment $\mathcal{Z}$ who controls an adversary $\mathcal{A}$ corrupting the receiver, and $\mathcal{Z}$ can distinguish between the two executions. We construct a distinguisher $\mathcal{D}$ for the leaky LPN assumption, as follows. $\mathcal{D}$ starts by simulating an execution of $\Pi^{\ell,s}_{\text{vole2k}}$, as in the above simulation, until it reaches the leakage queries in the $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$ instances. Here, $\mathcal{D}$ receives t queries $I_1, \dots, I_t \subset [n/t]$, forwards these to the leaky LPN challenger and uses its response to respond to $\mathcal{A}$. For the second set of leakage queries, it sends the sets $J_1, \dots, J_t$ to the challenger, and again uses its response to simulate the response to $\mathcal{A}$. If the LPN challenger aborts, the simulation is aborted. Finally, $\mathcal{D}$ receives the vector $\mathbf{y}_b$ from the LPN challenger, and uses its last $n - m$ entries to define the honest sender's output $\mathbf{x}$, which is given to $\mathcal{Z}$; the output $\mathbf{z}$ is defined to be $\Delta \cdot \mathbf{x} + \mathbf{y}$.

Notice that the way the leakage queries are simulated is identical to the ideal functionality $\mathcal{F}^{\ell,s}_{\text{sp-vole2k}}$. It follows that if $b = 0$ in the leaky LPN game, then the view of $\mathcal{Z}$ is identical to the real execution, while if $b = 1$, the view is the same as the ideal world. Therefore, the distinguishing advantage of $\mathcal{D}$ is the leaky LPN game is the same as that of $\mathcal{Z}$. □

Communication Complexity When we instantiate the single-point VOLE with our protocol $\Pi^{\ell,s}_{\text{sp-vole2k}}$ from Section 2.3, use the equality test sketched in Section 2.2.3, and Silent OT [BCG+19a, YWL+20, CRR21], our VOLE extension protocol $\Pi^{\ell,s}_{\text{vole2k}}$ with LPN parameters, (m, t, n) requires $m + 2t$ base VOLEs and $4t\ell + 2t\sigma + 4t\lceil \log n/t \rceil + (5 + 2\lceil \log n/t \rceil)t\kappa$ bit of communication. The costs for the single-point VOLE protocol are broken down in Section 2.3.3.

2.5 QuarkSilver: QuickSilver Modulo 2^k

We now construct the QuarkSilver zero-knowledge proof system, which is based on a similar principle as the QuickSilver protocol. The main technique to achieve soundness in QuickSilver [YSWW21a], similar to LPZK [DIO21a], is that a dishonest prover can only cheat in multiplication checks if it can come up with a quadratic polynomial of a certain form, which has a root Δ unknown to the prover. This is straightforward over fields, but over $\mathbb{Z}_{2^k}$ there might be many more than just two roots for a polynomial. Before constructing the zero-knowledge protocol, we therefore give upper-bounds on the number of roots of certain quadratic polynomials over $\mathbb{Z}_{2^k}$.

2.5.1 Bounded Solutions to Quadratic Equations in $\mathbb{Z}_{2^k}$

We examine the roots of the following polynomial modulo 2^ℓ:

$$f(x) = ax^2 + bx + c$$

We are only interested in the case where $a \neq 0 \mod 2^k$, while b and c may be chosen arbitrarily by the adversary. Finally, we also only look at roots $x \in \{0, \ldots, 2^s - 1\}$, since the secret MAC key Δ is sampled from this range.

Towards giving a bound, we will use the following basic fact about modular square roots.

Proposition 1. *Let $a \in \mathbb{Z}$ be an odd number. Then $x^2 = a \mod 2^\ell$ has at most 4 solutions.*

Proof of Proposition 1. Clearly, if there exists one solution x then there are 3 more solutions $-x, x + 2^{\ell-1}$ and $-x + 2^{\ell-1}$. We now show that these are the only such solutions.

For the sake of contradiction, let y be such that $y^2 = a \bmod 2^\ell$. Then

$$x^2 - y^2 = (x - y)(x + y) = 0 \bmod 2^\ell.$$

Both x, y must be odd as a is odd, hence both $x + y$ and $x - y$ are a multiple of 2. If $x+y = 0$ or $x-y = 0$ then $x = \pm y$. Assuming this is false, then there exist odd numbers f, g as well as positive i, j such that

$$x + y = f2^i \text{ , } x - y = g2^j \text{ and } i + j \geq \ell.$$

Since these equations hold over the integers, we additionally get that

$$(x + y) + (x - y) = 2x = f2^i + g2^j \Rightarrow x = f2^{i-1} + g2^{j-1}$$

where in the last step we divide over the integers. Since x must be odd, we have that either i or j must be 1. If $i = 1$ then

$$x - y = g2^{\ell-1} \Rightarrow x = y \bmod 2^{\ell-1}$$

whereas we get $x = -y \bmod 2^{\ell-1}$ if $j = 1$. □

We also use a version of Hensel's lemma (see e.g. [Hac07]), which allows lifting certain solutions to an equation modulo p up to solutions modulo p^ℓ.

Lemma 4 (Hensel's lemma)**.** *Let p be prime, $f(x)$ be a polynomial with integer coefficients and $f'(x)$ its derivative. If there exists an integer x^* such that*

$$f(x^*) = 0 \mod p^i \quad \textit{and} \quad f'(x^*) \neq 0 \mod p$$

then there is a unique integer y modulo p^{i+1} satisfying

$$f(y) = 0 \mod p^{i+1} \quad \textit{and} \quad x^* = y \mod p^i$$

Note that any solution to $f(x) = 0$ modulo p^ℓ is also a solution modulo p. Hence, if the derivatives of all the roots modulo p are non-zero, we are guaranteed that there are no more than two solutions modulo higher powers. The challenging case is when the derivative is zero. We now show the following.

Lemma 5. *Let $f(x) \in \mathbb{Z}[x]$ be a quadratic equation such that 2^r is the largest power of 2 dividing all coefficients. Then for any $\ell, s, s' \in \mathbb{N}$ such that $\ell - r > s'$ there are at most $2^{\max\{(2s-s')/2,1\}}$ solutions to $f(x) = 0 \mod 2^\ell$ in $\{0, \ldots, 2^s - 1\}$.*

Proof. First, we will divide $f(x)$ by 2^r, the largest power of two that divides all coefficients, then redefine f accordingly and solve:

$$f(x) = ax^2 + bx + c = 0 \mod 2^{\ell-r}$$

where now at least one of $\{a, b, c\}$ is odd.

Case 1: a and b are odd. We can use Lemma 4, since the derivative $f'(x) = 2ax + b$ is odd and, therefore, non-zero modulo 2. This means any solution modulo 2 lifts to a unique solution modulo higher powers, so there are at most 2 solutions modulo $2^{\ell-r}$.

Case 2: a is odd and b is even. Since a is invertible modulo $2^{\ell-r}$, we can complete the square: Define $g(y) := a \cdot y^2 + t$ with $t := c - b^2/4 \cdot (a^{-1} \mod 2^{\ell-r})$. Then we have $f(x) = g(y) \pmod{2^{\ell-r}}$ using the substitution $y = x + b/2 \cdot (a^{-1} \mod 2^{\ell-r})$:

$$\begin{aligned} g(y) &= a \cdot \left(x + b/2 \cdot (a^{-1} \bmod 2^{\ell-r})\right)^2 + c - b^2/4 \cdot (a^{-1} \bmod 2^{\ell-r}) \\ &= a \cdot x^2 + b \cdot x + b^2/4 \cdot (a^{-1} \bmod 2^{\ell-r}) + c - b^2/4 \cdot (a^{-1} \bmod 2^{\ell-r}) \\ &= ax^2 + bx + c = f(x) \pmod{2^{\ell-r}} \end{aligned}$$

So, to solve for x we can now solve $ay^2 = -t \mod 2^{\ell-r}$ for y, where the original constraint for x now puts y in the interval $\{b/2 \cdot (a^{-1} \bmod 2^{\ell-r}), \ldots, b/2 \cdot (a^{-1} \bmod 2^{\ell-r}) + 2^s - 1\}$. Since this substitution is just a constant shift, the maximal number of possible solutions in any interval of length 2^s for g directly translates into an upper bound on the number of solutions for f in the solution space.

Letting $t' = t \cdot (a^{-1} \bmod 2^{\ell-r})$, we now want to solve:

$$y^2 = -t' \pmod{2^{\ell-r}} \tag{2.2}$$

- **Case (2a): $t' = 0 \pmod{2^{\ell-r}}$.** Then, the solutions y are all the multiples of $2^{(\ell-r)/2}$. In an interval of length 2^s, there can be at most $2^s/2^{(\ell-r)/2} < 2^{s-s'/2}$ of these.

- **Case (2b): $t' \neq 0 \pmod{2^{\ell-r}}$.** Let $2^{v'}$ be the largest power of two dividing t'. Since $-t'$ is a square, v' must be even so we can write $v' = 2v$ for some

$v \leq (\ell - r - 1)/2$. Write $-t' = u \cdot 2^{2v}$ for some odd $u \in \mathbb{Z}$, and let $z = y/2^v$, so we have

$$z^2 = u \pmod{2^{\ell-r-2v}}. \tag{2.3}$$

Any solution y for (2.2) satisfies $y = z \cdot 2^v \bmod 2^{\ell-r-2v}$ for some z that is a solution to (2.3). So, there is a $k \in \mathbb{Z}$ such that

$$\begin{aligned} & y = z \cdot 2^v + k \cdot 2^{\ell-r-2v} \\ \Rightarrow\ & y^2 = z^2 \cdot 2^{2v} + 2 \cdot z \cdot k \cdot 2^{\ell-r-v} + k^2 \cdot 2^{2(\ell-r-2v)} \\ \Rightarrow\ & y^2 = -t' + z \cdot k \cdot 2^{\ell-r-v+1} \pmod{2^{\ell-r}}. \end{aligned} \tag{2.4}$$

To bound the number of solutions y, it suffices to bound the number of z and k satisfying the above. For the y's to be distinct mod $2^{\ell-r}$, we need $k < 2^{2v}$, which means there are 2^{v+1} possibilities for k, given by $k = i \cdot 2^{v-1}$ for all $i \in \{0, \ldots, 2^{v+1} - 1\}$. Since z is odd, from Proposition 1 there are no more than 4 solutions to (2.3), which are of the form $\pm z_0$ and $2^{\ell-r-2v-1} \pm z_0$ for some z_0. However, it is easy to see that plugging $z := z_0 + 2^{\ell-r-2v-1}$ into (2.4) gives the same set of solutions for y as with $z := z_0$, so we only need to count $\pm z_0$. Overall, this shows there are at most 2^{v+2} solutions y to (2.4).

Since each solution in the set defined in Equation (2.4) (with $k = i \cdot 2^{v-1}$) is spaced apart by $2^{\ell-r-v-1}$, any interval of size 2^s contains no more than $2^{s-\ell+r+v+1}$ of these. From the fact that $v \leq (\ell - r - 1)/2$, we get

$$2^{s-\ell+r+v+1} \leq 2^{(\ell-r-1)/2-\ell+r+s+1)} = 2^{(-(\ell-r)+1+2s)/2} \leq 2^{(2s-s')/2}$$

as required (where the last inequality holds since $\ell - r > s'$).

Case 3: a is even, b is odd. In this case, $f(x) = x + c \mod 2$ is linear, hence, the unique solution $x = c \bmod 2$ gives a unique solution modulo $2^{\ell-r}$ via Lemma 4.

Case 4: a, b are even, c is odd. Here, $f(x) = 0$ has no solutions modulo 2, so also no solutions modulo any higher power. □

2.5.2 Bounded Solutions for a Generalized Setting

In the previous subsection, we analyzed the setting that one would end up with when constructing a soundness argument for our check for one multiplication. In order to amortize this check to t multiplications, we generalize the security game in the following theorem.

Theorem 5. *Let $\ell, s, k \in \mathbb{N}^+$ so that $\ell \geq k + 2s$ and consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:*

1. *$\mathcal{C}$ chooses $\Delta \in \mathbb{Z}_{2^s}$ uniformly at random.*

2. *$\mathcal{A}$ sends $\delta_0, \dots, \delta_t \in \mathbb{Z}$ such that not all δ_i for $i > 0$ are $0 \bmod 2^k$.*

3. *$\mathcal{C}$ chooses $\chi_1, \dots, \chi_t \leftarrow \mathbb{Z}_{2^s}$ uniformly at random and sends these to $\mathcal{A}$.*

4. *$\mathcal{A}$ sends $b, c \in \mathbb{Z}$.*

5. *$\mathcal{A}$ wins iff $(\delta_0 + \sum_i \chi_i \delta_i)\Delta^2 + b\Delta + c = 0 \bmod 2^\ell$.*

Then $\mathcal{A}$ can win with probability at most $(\ell - k + 2) \cdot 2^{-s+1}$.

The proof of Theorem 5 follows a similar way as Lemma 1 of [CDE+18]. The key observation is that Step 3 determines an upper-bound on r, the largest number such that 2^r divides all coefficients of the polynomial. This is because no choice of b, c can increase r as it also must divide the leading coefficient, which is randomized. By the random choice of the χ_i, one can show that the larger r is, the smaller the chance that it divides $\delta_0 + \sum_i \chi_i \delta_i$. Since a larger r leads to more roots of the polynomial, we can then bound the overall attack success for each possible r.

In the proof of Theorem 5, we will use the following statement.

Proposition 2. *Let $\ell, k, s \in \mathbb{N}^+$ so that $\ell \geq k + s$, and $\delta_0, \dots, \delta_t \in \mathbb{Z}$ be values such that not all δ_i for $i > 0$ are $0 \bmod 2^k$. Then for any $j \in \{0, \dots, \ell - k\}$*

$$\Pr\left[\delta_0 + \sum_{i \in [t]} \chi_i \cdot \delta_i = 0 \bmod 2^{k+j} \;\middle|\; \chi_1, \dots, \chi_t \leftarrow \mathbb{Z}_{2^s}\right] \leq 2^{-\min(j,s)}.$$

The proof is an adaptation of the proof of Part *iii* of Lemma 1 of [CDE+18].

Proof of Proposition 2. Let $j \in \{0, \dots, \ell - k\}$ be arbitrary. Without loss of generality assume that $\delta_t \neq 0 \pmod{2^k}$. Let $v \in \mathbb{N}$ be maximal such that $2^v \mid \delta_t$. This implies $v < k$. Define $S := \delta_0 + \sum_{i \in [t]} \chi_i \cdot \delta_i$ and $S' := -\delta_0 - \sum_{i \in [t-1]} \chi_i \cdot \delta_i$, and let $W := \min(\ell, e)$ where $e \in \mathbb{N}$ is maximal such that $2^e \mid S$.

Suppose $W = k + j$. By definition, $2^W \mid S$ which is equivalent to $S = 0 \pmod{2^{k+j}}$. Rewrite the equation as $\chi_t \cdot \delta_t = S' \pmod{2^{k+j}}$. By the definition of v, both sides must be multiples of 2^v. So we can divide by 2^v over the integers and reduce the modulus accordingly. Then $\delta/(2^v)$ is odd and, thus, invertible modulo 2^{k+j-v}. Since $v < k$, we have $k + j - v > j$, and can reduce the modulus further to 2^j:

$$\chi_t = \frac{S'}{2^v} \cdot \left(\frac{\delta_t}{2^v}\right)^{-1} \pmod{2^j}. \tag{2.5}$$

The left-hand side χ_t is distributed uniformly at random in $\mathbb{Z}_{2^s}$ and independent of the right-hand side. Hence, if $j < s$, then Equation (2.5) holds with probability at most 2^{-j}. For $j \geq s$, it holds with probability at most 2^{-s}. The proposition follows. ☐

Proof of Theorem 5. Let $a = \delta_0 + \sum_i \chi_i \delta_i \bmod 2^\ell$, then $\mathcal{A}$ wins iff $f(\Delta) = a\Delta^2 + b\Delta + c = 0 \bmod 2^\ell$. Let r be the largest value such that 2^r divides all a, b, c. We have that

$$\begin{aligned}
\Pr[\mathcal{A} \text{ wins}] \leq & \sum_{i=0}^{\ell} \Pr[\mathcal{A} \text{ wins}, r = i] \\
= & \Pr[\mathcal{A} \text{ wins} \mid r \in \{0, \dots, k-1\}] \cdot \Pr[r \in \{0, \dots, k-1\}] \\
& + \sum_{i=k}^{\ell} \Pr[\mathcal{A} \text{ wins}, r = i] \\
\leq & \Pr[\mathcal{A} \text{ wins} \mid r \in \{0, \dots, k-1\}] + \sum_{i=k}^{\ell} \Pr[\mathcal{A} \text{ wins}, r = i] \\
\leq & 2^{-\min\{(\ell-k)/2, s-1\}} + \sum_{j=0}^{\ell-k} \Pr[\mathcal{A} \text{ wins}, r = k + j]
\end{aligned} \tag{2.6}$$

Here, in the last step we use Lemma 5 where we set $\ell := \ell$ and $s' := \ell - k$.

By definition of r, if $r = k + j$ and $\mathcal{A}$ wins, then 2^{k+j} must divide a. Therefore

$$\begin{aligned}
& \Pr[\mathcal{A} \text{ wins}, r = k + j] \\
= & \Pr[\mathcal{A} \text{ wins}, r = k + j, 2^{k+j} \text{ divides } a] \\
= & \Pr[\mathcal{A} \text{ wins}, r = k + j \mid 2^{k+j} \text{ divides } a] \cdot \Pr[2^{k+j} \text{ divides } a].
\end{aligned}$$

Claim 3. *For $j \in \{0, \dots, \ell - k\}$, the following inequalities holds (with $\lambda := \ell - k - s$):*

a) $\Pr[2^{k+j} \textit{ divides } a] \leq 2^{-\min\{j,s\}}$,

b) $\Pr[\mathcal{A} \textit{ wins}, r = k + j \mid 2^{k+j} \textit{ divides } a] \leq 2^{-\min\{(s+\lambda-j)/2, s\}+1}$,

c) $\Pr[\mathcal{A} \textit{ wins}, r = k + j] \leq 2^{-s+1}$.

Proof of Claim.

a) Follows directly from Proposition 2.

b) By Lemma 5 for $\ell := \ell$, $r := k + j$, $s' := \ell - r - 1 = s + \lambda - j - 1$ and any $j \in \{0, \dots, \ell - k\}$, there are at most $2^{\max\{(2s-s')/2, 1\}} \leq 2^{\max\{(s+j-\lambda)/2, 0\}+1}$ solutions in the range $\{0, \dots, 2^s - 1\}$ to the equation $f(x) = 0 \pmod{2^\ell}$. Since there are 2^s choices for Δ, this means that

$$\begin{aligned}
\Pr[\mathcal{A} \text{ wins}, r = k + j \mid 2^{k+j} \text{ divides } a] \leq & \frac{2^{\max\{(s+j-\lambda)/2, 0\}+1}}{2^s} \\
= 2^{\max\{(-s+j-\lambda)/2, -s\}+1} = & 2^{-\min\{(s+\lambda-j)/2, s\}+1}.
\end{aligned}$$

c) Combining the previous two parts, we obtain

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ wins}, r = k + j] \\
&= \Pr[\mathcal{A} \text{ wins}, r = k + j \mid 2^{k+j} \text{ divides } a] \cdot \Pr[2^{k+j} \text{ divides } a] \\
&\leq 2^{-\min\{(s+\lambda-j)/2, s\}+1} \cdot 2^{-\min\{j,s\}} \\
&= 2^{-\min\{(s+\lambda-j)/2, s\}+1-\min\{j,s\}}.
\end{aligned}$$

For $j \in \{0, \dots, s\}$, this can be simplified to

$$\begin{aligned}
\Pr[\mathcal{A} \text{ wins}, r = k + j] &= 2^{-\min\{(s+\lambda-j)/2, s\}+1-j} \\
&= 2^{-\min\{(s+\lambda+j)/2, s+j\}+1} \\
&\overset{(\star)}{\leq} 2^{-\min\{(2s+j)/2, s+j\}+1} \\
&= 2^{-\min\{s+j/2, s+j\}+1} \\
&= 2^{-s-j/2+1} \\
&\leq 2^{-s+1},
\end{aligned}$$

where we use the fact that $\lambda \geq s$ at step $(\star)$, which follows from the assumption $\ell - k \geq 2s$ and the definition of λ. For $j \in \{s, \dots, \ell - k\}$, we obtain the bound directly from Part a):

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ wins}, r = k + j] \\
&= \Pr[\mathcal{A} \text{ wins}, r = k + j \mid 2^{k+j} \text{ divides } a] \cdot \Pr[2^{k+j} \text{ divides } a] \\
&\leq \Pr[\mathcal{A} \text{ wins}, r = k + j \mid 2^{k+j} \text{ divides } a] \cdot 2^{-\min\{j,s\}} \\
&\leq 2^{-\min\{j,s\}} = 2^{-s} \leq 2^{-s+1}.
\end{aligned}$$

■

Continuing from Equation (2.6), we use the Claim proved above and the fact that $\ell \geq k + 2s$ to get

$$\begin{aligned}
\Pr[\mathcal{A} \text{ wins}] &\overset{(2.6)}{\leq} 2^{-\min\{(\ell-k)/2, s-1\}} + \sum_{j=0}^{\ell-k} \Pr[\mathcal{A} \text{ wins}, r = k + j] \\
&\leq 2^{-\min\{s, s-1\}} + \sum_{j=0}^{\ell-k} 2^{-s+1} \\
&\leq 2^{-s+1} + \sum_{j=0}^{\ell-k} 2^{-s+1} \\
&\leq (\ell - k + 2) \cdot 2^{-s+1}.
\end{aligned}$$

□

Corollary 1. *Let $\sigma \geq 7$ be a statistical security parameter. By setting $s := \sigma + \log\sigma + 3$ and $\ell := k + 2s$, any adversary $\mathcal{A}$ can win the game from Theorem 5 with probability at most $2^{-\sigma}$.*

Proof of Corollary 1. Plugging in the values of s and ℓ into the winning probability stated in the theorem gives us:

$$\begin{aligned}\Pr[\mathcal{A} \text{ wins}] &\leq (\ell - k + 2) \cdot 2^{-s+1} \\ &= (k + 2s - k + 2) \cdot 2^{-s+1} \\ &= (2s + 2) \cdot 2^{-s+1} \\ &= (2\sigma + 2\log\sigma + 8) \cdot 2^{-\sigma-\log\sigma-2}.\end{aligned}$$

By taking the logarithm of both sides, we get

$$\begin{aligned}\log\Pr[\mathcal{A} \text{ wins}] &\leq \log(2\sigma + 2\log(\sigma) + 8) - \sigma - \log(\sigma) - 2 \\ &\overset{(\star)}{\leq} \log(4\sigma) - \sigma - \log(\sigma) - 2 \\ &= \log(\sigma) + 2 - \sigma - \log(\sigma) - 2 \\ &= -\sigma,\end{aligned}$$

where we use that $2\sigma + 2\log(\sigma) + 8 \leq 4\sigma$ $(\star)$ holds for all $\sigma \geq 7$. Hence, we have bounded the winning probability as $\Pr[\mathcal{A} \text{ wins}] \leq 2^{-\sigma}$. □

2.5.3 QuarkSilver

We now construct the QuarkSilver zero-knowledge proof system. Its main building block are linearly homomorphic commitments instantiated from VOLEs over $\mathbb{Z}_{2^\ell}$.

Linearly Homomorphic Commitments.

As in the A2B [BBMH+21b] zero-knowledge protocols, we use linearly homomorphic commitments from VOLE to authenticate values in $\mathbb{Z}_{2^k}$: Define a commitment $[x]$ to a value $x \in \mathbb{Z}_{2^k}$ known to the prover by a global key $\Delta \in_R \mathbb{Z}_{2^s}$ and values $\mathsf{K}[x], \mathsf{M}[x] \in_R \mathbb{Z}_{2^\ell}$ with $\ell \geq k + s$ so that

$$\mathsf{K}[x] = \mathsf{M}[x] + \widetilde{x} \cdot \Delta \pmod{2^\ell} \tag{2.7}$$

holds for $\widetilde{x} = x \pmod{2^k}$. Here the prover knows $\widetilde{x}$ and $\mathsf{M}[x]$, and the verifier knows Δ and $\mathsf{K}[x]$. To open the commitment, the prover reveals $\widetilde{x}, \mathsf{K}[x]$ to the verifier who checks that the aforementioned equalities hold.

The commitment scheme is linearly homomorphic, as no interaction is needed to compute $[a \cdot x + b]$ from $[x]$ for publicly known $a, b \in \mathbb{Z}_{2^k}$: $\mathcal{P}, \mathcal{V}$ simply update $\widetilde{x}, \mathsf{K}[x]$ and $\mathsf{M}[x]$ in the appropriate way modulo 2^ℓ. The same linearity also holds when adding commitments. Unfortunately, the upper $\ell - k$ bits of $\widetilde{x}$ may not be uniformly random when opening a commitment. To resolve this, the prover instead opens $[x + 2^k y]$ using a random commitment $[y]$.

QuarkSilver Π^k_{QS}

The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ have agreed on a circuit $\mathcal{C}$ over $\mathbb{Z}_{2^k}$ with n inputs and t multiplication gates, and $\mathcal{P}$ holds a witness $\mathbf{w} \in \mathbb{Z}_{2^k}^n$ so that $\mathcal{C}(\mathbf{w}) = 1$.

Preprocessing phase The preprocessing phase is independent of $\mathcal{C}$ and just needs upper bounds on the number of inputs and multiplication gates of $\mathcal{C}$ as input.

1. $\mathcal{P}$ and $\mathcal{V}$ send (Init) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, and $\mathcal{V}$ receives $\Delta \in \mathbb{Z}_{2^s}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send (Extend, n + t + 2) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, which returns authenticated values $([\mu_i])_{i\in[n]}$, $([\nu_i])_{i\in[t]}$, $[o]$, and $[\pi]$, where all $\widetilde{\mu}_i, \widetilde{\nu}_i, \widetilde{o}, \widetilde{\pi} \in_R \mathbb{Z}_{2^\ell}$.

Online phase

1. For each input w_i, $i \in [n]$, $\mathcal{P}$ sends $\delta_i := w_i - \widetilde{\mu}_i$ to $\mathcal{V}$, and both parties locally compute $[w_i] := [\mu_i] + \delta_i$.
2. For each gate $(\alpha, \beta, \gamma, T) \in \mathcal{C}$, in topological order:
   - If $T = \mathsf{Add}$, then $\mathcal{P}$ and $\mathcal{V}$ locally compute $[w_\gamma] := [w_\alpha] + [w_\beta]$.
   - If $T = \mathsf{Mul}$ and this is the ith multiplication gate, then $\mathcal{P}$ sends $d_i := w_\alpha \cdot w_\beta - \widetilde{\nu}_i$, and both parties locally compute $[w_\gamma] := [\nu_i] + d_i$.
3. For the ith multiplication gate, the parties hold $([w_\alpha], [w_\beta], [w_\gamma])$ with $\mathsf{K}[w_i] = \mathsf{M}[w_i] + \widetilde{w}_i \cdot \Delta$ for $i \in \{\alpha, \beta, \gamma\}$.
   - $\mathcal{P}$ computes $A_{0,i} := \mathsf{M}[w_\alpha] \cdot \mathsf{M}[w_\beta] \in \mathbb{Z}_{2^\ell}$ and $A_{1,i} := \widetilde{w}_\alpha \cdot \mathsf{M}[w_\beta] + \widetilde{w}_\beta \cdot \mathsf{M}[w_\alpha] - \mathsf{M}[w_\gamma] \in \mathbb{Z}_{2^\ell}$.
   - $\mathcal{V}$ computes $B_i := \mathsf{K}[w_\alpha] \cdot \mathsf{K}[w_\beta] - \Delta \cdot \mathsf{K}[w_\gamma] \in \mathbb{Z}_{2^\ell}$.
4. $\mathcal{P}$ and $\mathcal{V}$ run the following check:
   (a) Set $A_0^* := \mathsf{M}[o]$, $A_1^* := \widetilde{o}$, and $B^* := \mathsf{K}[o]$ so that $B^* = A_0^* + A_1^* \cdot \Delta$.
   (b) $\mathcal{V}$ samples $\boldsymbol{\chi} \in_R \mathbb{Z}_{2^s}^t$ and sends it to $\mathcal{P}$.
   (c) $\mathcal{P}$ computes $U := \sum_{i\in[t]} \chi_i \cdot A_{0,i} + A_0^* \in \mathbb{Z}_{2^\ell}$ and $V := \sum_{i\in[t]} \chi_i \cdot A_{1,i} + A_1^* \in \mathbb{Z}_{2^\ell}$, and sends (U, V) to $\mathcal{V}$.
   (d) $\mathcal{V}$ computes $W := \sum_{i\in[t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$, and checks that $W = U + V \cdot \Delta \pmod{2^\ell}$. If the check fails, $\mathcal{V}$ outputs false and aborts.
5. For the single output wire w_h, both parties hold $[w_h]$. They first compute $[z] := [w_h] + 2^k \cdot [\pi]$. Then $\mathcal{P}$ sends $\widetilde{z}$ and $\mathsf{M}[z]$ to $\mathcal{V}$ who checks that $\widetilde{z} = 1 \pmod{2^k}$ and $\mathsf{K}[z] = \mathsf{M}[z] + \widetilde{z} \cdot \Delta$. $\mathcal{V}$ outputs true iff the check passes, and false otherwise.

Figure 2.7: Zero-knowledge protocol for circuit satisfiability in the $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$-hybrid model with $s := \sigma + \log(\sigma) + 3$ and $\ell := k + 2s$ for statistical security parameter σ.

How QuarkSilver Works.

QuarkSilver follows the established commit-and-prove paradigm for zero-knowledge proofs. For the commitments, we use the linearly homomorphic commitments described above. For a circuit with n inputs and t multiplications, we start by generating $n+t+2$ authenticated random values $[r_1], \dots, [r_{n+t+2}]$ with $\widetilde{r}_i \in_R \mathbb{Z}_{2^\ell}$ for $i \in [n+t+2]$, i.e. commitments to random values. For this, $\mathcal{P}$ and $\mathcal{V}$ call $(\mathsf{Extend}, n+t+2)$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. $\mathcal{P}$ then commits to $\mathbf{w}$ using the first n random commitments. Next, the parties evaluate the circuit topologically, computing commitments to the outputs of linear gates using the homomorphism of $[\cdot]$. For each multiplication gate, $\mathcal{P}$ commits to the output using another unused random commitment. It then remains to show that the commitment to the output of the circuit is a commitment to 1 and that all commited outputs of muliplication gates are indeed consistent with the committed inputs.

To verify the committed output wire, QuarkSilver uses the "blinded opening" procedure that was introduced above. This procedure will consume another random commitment. To check validity of a multiplication, observe that for 3 commitments $[w_\alpha], [w_\beta], [w_\gamma]$ with $\gamma = \alpha \cdot \beta \bmod 2^k$ it holds that

$$\underbrace{\mathsf{K}[w_\alpha] \cdot \mathsf{K}[w_\beta] - \Delta \cdot \mathsf{K}[w_\gamma]}_{B} =$$

$$\underbrace{\mathsf{M}[w_\alpha] \cdot \mathsf{M}[w_\beta]}_{A_0} + \Delta \cdot \underbrace{(\tilde{w}_\alpha \cdot \mathsf{M}[w_\beta] + \tilde{w}_\beta \cdot \mathsf{M}[w_\alpha] - \mathsf{M}[w_\gamma])}_{A_1},$$

where $\mathcal{P}$ can compute A_0, A_1 while $\mathcal{V}$ can compute B. Hence, by sending A_0, A_1 to $\mathcal{V}$ the latter can check that the relation on B, Δ holds. Instead of sending these for every multiplication, we check all t relations simultaneously by having $\mathcal{V}$ choose a string $\chi \leftarrow \mathbb{Z}^t_{2^s}$, so that the prover instead sends $(\sum_i \chi_i A_{0,i}, \sum_i \chi_i A_{1,i})$ while the verifier checks the relation on $\sum_i \chi_i B_i$ and Δ. Since revealing these linear combinations directly might leak information, $\mathcal{P}$ will first blind the opening with the remaining random commitment from the preprocessing.

While the completeness and zero-knowledge of the aforementioned protocol follows directly, we will explain the soundness in more detail in the security proof. The full protocol is presented in Figure 2.7.

Security of the QuarkSilver Protocol

Theorem 6. *The protocol Π^k_{QS} (Figure 2.7) securely realizes the functionality $\mathcal{F}^k_{\mathsf{ZK}}$ in the $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$-hybrid model when instantiated with the parameters $s := \sigma + \log(\sigma) + 3$ and $\ell := k + 2s$: No unbounded environment $\mathcal{Z}$ can distinguish the real execution of the protocol from a simulated one except with probability $2^{-\sigma+1}$.*

As our protocol is an adaption of QuickSilver [YSWW21a], the structure of our proof is also similar. The main difference, lies in the proof of soundness of the multiplication check.

Proof of Theorem 6. We divide the proof of security into two parts. First we cover the case of a corrupted prover, then that of a corrupted verifier. In each case we define a PPT simulator $\mathcal{S}$.

Corrupted Prover: The simulation is set up as follows: $\mathcal{S}$ simulates a party $\mathcal{P}^*$ in its head and gives control to $\mathcal{Z}$, and sends $(\mathsf{corrupt}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{ZK}}^k$. It also simulates an instance of $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$. We assume that the circuit $\mathcal{C}$ is known.

1. Simulation of the preprocessing phase: $\mathcal{S}$ simulates the (Init) and $(\mathsf{Extend}, n+t+2)$ calls to $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$. For (Init), it samples $\Delta \in_R \mathbb{Z}_{2^s}$. Since $\mathcal{P}^*$ acts as the sender $\mathcal{S}$ towards $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$, it is allowed to choose the sender's output of the (Extend) call. Hence, $\mathcal{S}$ receives $(\widetilde{\mu}_i, \mathsf{M}[\mu_i]), (\widetilde{\nu}_j, \mathsf{M}[\nu_j]), (\widetilde{o}, \mathsf{M}[o]), (\widetilde{\pi}, \mathsf{M}[\pi]) \in \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$ for $i \in [n]$ and $j \in [t]$ from $\mathcal{P}^*$. Then $\mathcal{S}$ can compute matching values $\mathsf{K}[\mu_i], \mathsf{K}[\nu_j], \mathsf{K}[o], \mathsf{K}[\pi] \in \mathbb{Z}_{2^\ell}$ according to Equation (2.7).

2. To simulate the online phase, $\mathcal{S}$ executes the steps of $\mathcal{V}$ while also keeping track of $\mathcal{P}^*$'s wire values:

   - For every circuit input $i \in [n]$ it receives $\delta_i \in \mathbb{Z}_{2^\ell}$ from $\mathcal{P}^*$ and computes $\mathcal{V}$'s part of $[w_i] := [\mu_i] + \delta_i$, as well as $\widetilde{w}_i := \widetilde{\mu}_i + \delta_i \in \mathbb{Z}_{2^\ell}$.
   - For every addition gate (α, β, γ) it computes $\mathcal{V}$'s part of $[w_\gamma] := [w_\alpha] + [w_\beta]$, as well as $\widetilde{w}_\gamma := \widetilde{w}_\alpha + \widetilde{w}_\beta \in \mathbb{Z}_{2^\ell}$ and $\mathsf{M}[w_\gamma] := \mathsf{M}[w_\alpha] + \mathsf{M}[w_\beta]$.
   - For the ith multiplication gate (α, β, γ) it receives $d_i \in \mathbb{Z}_{2^\ell}$ from $\mathcal{P}^*$ and computes $\mathcal{V}$'s part of $[w_\gamma] := [\nu_i] + d_i$, as well as $\widetilde{w}_\gamma := \widetilde{\nu}_i + d_i \in \mathbb{Z}_{2^\ell}$. Additionally it computes $B_i := \mathsf{K}[w_\alpha] \cdot \mathsf{K}[w_\beta] + \Delta \cdot \mathsf{K}[w_\gamma]$.
   - $\mathcal{S}$ sends $\boldsymbol{\chi} \in_R \mathbb{Z}_{2^s}^t$ to $\mathcal{P}^*$ and receives two values $U', V' \in \mathbb{Z}_{2^\ell}$ as response. It computes $W := \sum_{i \in [t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$ where $B^* := \mathsf{K}[o]$.
   - If $W \neq U' + V' \cdot \Delta \pmod{2^\ell}$, then $\mathcal{S}$ sends $(\mathsf{Prove}, \mathcal{C}, \bot)$ on behalf of $\mathcal{P}^*$ to $\mathcal{F}_{\mathsf{ZK}}^k$ and simulates an aborting $\mathcal{V}$.
   - For the single output wire w_h, $\mathcal{S}$ already holds $\mathsf{K}[w_h]$ and computes $\mathsf{K}[z] := \mathsf{K}[w_h] + 2^k \cdot \mathsf{K}[\pi]$. Then it receives two values $\widetilde{z}, \mathsf{M}[z] \in \mathbb{Z}_{2^\ell}$ from $\mathcal{P}^*$ and checks if $\mathsf{K}[z] \stackrel{?}{=} \mathsf{M}[z] + \widetilde{z} \cdot \Delta$ holds and $\widetilde{z} = 1 \pmod{2^k}$. If this is the case, then $\mathcal{S}$ sends $(\mathsf{Prove}, \mathcal{C}, \mathbf{w})$ with $\mathbf{w} := (\widetilde{w}_i \bmod 2^k)_{i \in [n]}$ on behalf of $\mathcal{P}$ to $\mathcal{F}_{\mathsf{ZK}}^k$.

Since $\mathcal{S}$ behaves like an honest $\mathcal{V}$ towards $\mathcal{P}^*$, the view of $\mathcal{P}$ in the simulation perfectly matches its view in the real protocol. Now we need to show that $\mathcal{V}$ outputs the same in the simulation and in the real execution, except with negligible probability. If the honest $\mathcal{V}$ rejects the proof in the protocol, then also $\mathcal{V}$'s output of the ideal $\mathcal{F}_{\mathsf{ZK}}^k$ is false. Now we need to bound the probability that the honest $\mathcal{V}$ accepts the proof, but the ideal $\mathcal{F}_{\mathsf{ZK}}^k$ still outputs false, i.e., $\mathcal{P}^*$ has successfully fooled $\mathcal{V}$ into accepting a proof of a wrong statement.

A corrupted $\mathcal{P}^*$ not knowing a valid witness for the circuit can try to cheat in two ways: It can try to circumvent the multiplication check in Step 4 or it can try to

open $[z]$ in Step 5 to an invalid value. The latter succeeds with probability at most 2^{-s} [BBMH+21b].

Now we consider the former case: For the ith multiplication gate (α, β, γ), let $\widetilde{w}_\gamma = \widetilde{w}_\alpha \cdot \widetilde{w}_\beta + e_i \pmod{2^\ell}$, where $\widetilde{w}_\alpha, \widetilde{w}_\beta, \widetilde{w}_\gamma \in \mathbb{Z}_{2^\ell}$ are the values contained in the commitments $[w_\alpha], [w_\beta], [w_\gamma]$ and $e_i \in \mathbb{Z}_{2^\ell}$ is a possible error. Suppose that not all $e_i = 0 \pmod{2^k}$ for $i \in [t]$, i.e., there is an error in the lower k bits that we care about.

Then we have (all over $\mathbb{Z}_{2^\ell}$)

$$\begin{aligned}
\mathsf{K}[w_\gamma] &= \mathsf{M}[w_\gamma] + \widetilde{w}_\gamma \cdot \Delta \\
&= \mathsf{M}[w_\gamma] + (\widetilde{w}_\alpha \cdot \widetilde{w}_\beta + e_i) \cdot \Delta \\
&= \mathsf{M}[w_\gamma] + (\widetilde{w}_\alpha \cdot \widetilde{w}_\beta) \cdot \Delta + e_i \cdot \Delta,
\end{aligned}$$

and

$$\begin{aligned}
B_i &= \mathsf{K}[w_\alpha] \cdot \mathsf{K}[w_\beta] - \Delta \cdot \mathsf{K}[w_\gamma] \\
&= (\mathsf{M}[w_\alpha] + \widetilde{w}_\alpha \cdot \Delta) \cdot (\mathsf{M}[w_\beta] + \widetilde{w}_\beta \cdot \Delta) - \Delta \cdot (\mathsf{M}[w_\gamma] + \widetilde{w}_\gamma \cdot \Delta) \\
&= (\mathsf{M}[w_\alpha] + \widetilde{w}_\alpha \cdot \Delta) \cdot (\mathsf{M}[w_\beta] + \widetilde{w}_\beta \cdot \Delta) - \Delta \cdot (\mathsf{M}[w_\gamma] + (\widetilde{w}_\alpha \cdot \widetilde{w}_\beta + e_i) \cdot \Delta) \\
&= (\mathsf{M}[w_\alpha] \cdot \mathsf{M}[w_\beta]) + (\widetilde{w}_\alpha \cdot \mathsf{M}[w_\beta] + \mathsf{M}[w_\alpha] \cdot \widetilde{w}_\beta - \mathsf{M}[w_\gamma]) \cdot \Delta - e_i \cdot \Delta^2 \\
&= A_{i,0} + A_{i,1} \cdot \Delta - e_i \cdot \Delta^2,
\end{aligned}$$

where $A_{i,0}$ and $A_{i,1}$ denote the values that an honest $\mathcal{P}$ would compute. So now B_i is the result of evaluating a quadratic polynomial at Δ instead of a linear one. The equations for all gates are aggregated using the random linear combination:

$$\begin{aligned}
W &= \sum_{i \in [t]} \chi_i \cdot B_i + B^* \\
&= \sum_{i \in [t]} \chi_i \cdot \left(A_{i,0} + A_{i,1} \cdot \Delta - e_i \cdot \Delta^2\right) + A_0^* + A_1^* \cdot \Delta \\
&= \left(\sum_{i \in [t]} \chi_i \cdot A_{i,0} + A_0^*\right) + \left(\sum_{i \in [t]} \chi_i \cdot A_{i,1} + A_1^*\right) \cdot \Delta - \left(\sum_{i \in [t]} \chi_i \cdot e_i\right) \cdot \Delta^2 \\
&= U + V \cdot \Delta - \left(\sum_{i \in [t]} \chi_i \cdot e_i\right) \cdot \Delta^2
\end{aligned} \tag{2.8}$$

Here, U, V denote the values that an honest $\mathcal{P}$ would send to $\mathcal{V}$. The corrupted $\mathcal{P}^*$ may choose to send some values $U' := U + e_U$ and $V' := V + e_V$ instead. Note that $\mathcal{V}$ accepts the proof if $W = U' + V' \cdot \Delta$ holds. By subtracting Equation (2.8) from this, we get that $\mathcal{V}$ accepts if

$$0 = e_U + e_V \cdot \Delta + \left(\sum_{i \in [t]} \chi_i \cdot e_i\right) \cdot \Delta^2 \pmod{2^\ell}. \tag{2.9}$$

holds.

The steps in the protocol corresponding exactly to the game defined in Theorem 5: Initially, $\Delta \in_R \mathbb{Z}_{2^s}$ is sampled. When committing to the results of the multiplications, $\mathcal{P}^*$ defines the error values $e_1, \ldots, e_t \in \mathbb{Z}_{2^\ell}$ where not all of the e_i are 0 modulo 2^k if $\mathcal{P}^*$ tries to cheat. After $\mathcal{P}^*$ has committed itself on the e_i, $\mathcal{V}$ samples the coefficients $\chi_1, \ldots, \chi_t \in_R \mathbb{Z}_{2^s}$ of the random linear combination. Finally, the prover responds by choosing values $e_U, e_V \in \mathbb{Z}_{2^\ell}$, and wins the game, i.e., cheats successfully, if Equation (2.9) holds. Hence, we can apply Corollary 1 to bound the probability that this happens with $2^{-\sigma}$.

By the union bound, no adversary can break the soundness of the protocol except with probability at most $2^{-s} + 2^{-\sigma} \leq 2^{-\sigma+1}$.

Corrupted Verifier: The simulation of the verifier's view is straightforward: $\mathcal{S}$ simulates a party $\mathcal{V}^*$ in its head and gives control to $\mathcal{Z}$, and sends $(\mathsf{corrupt}, \mathcal{V})$ to $\mathcal{F}^k_{\mathsf{ZK}}$. It also simulates an instance of $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. We assume that the circuit $\mathcal{C}$ to prove is known.

1. Simulation of the preprocessing phase: $\mathcal{S}$ simulates the (Init) and $(\mathsf{Extend}, n+t+2)$ calls to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. Since $\mathcal{V}^*$ acts as the sender $\mathcal{S}$ towards $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, it is allowed to choose its outputs. For (Init), $\mathcal{S}$ receives $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{V}^*$, and $\mathsf{K}[\mu_i], \mathsf{K}[\nu_j], \mathsf{K}[o], \mathsf{K}[\pi] \in \mathbb{Z}_{2^\ell}$ for $i \in [n]$ and $j \in [t]$ from $\mathcal{V}^*$.

2. To simulate the online phase, $\mathcal{S}$ proceeds as follows:

   - For every circuit input $i \in [n]$ it sends a random $\delta_i \in_R \mathbb{Z}_{2^\ell}$ to $\mathcal{V}^*$, and computes $\mathsf{K}[w_i] := \mathsf{K}[\mu_i] - \delta_i \cdot \Delta$.
   - For the ith multiplication gate (α, β, γ) it sends a random $d_i \in_R \mathbb{Z}_{2^\ell}$ to $\mathcal{V}^*$, and computes $\mathsf{K}[w_\gamma] := \mathsf{K}[\nu_i] - d_i \cdot \Delta$ and $B_i := \mathsf{K}[w_\alpha] \cdot \mathsf{K}[w_\beta] + \Delta \cdot \mathsf{K}[w_\gamma] \in \mathbb{Z}_{2^\ell}$ as the honest $\mathcal{V}$ would do.
   - $\mathcal{S}$ receives $\boldsymbol{\chi} \in \mathbb{Z}^t_{2^s}$ from $\mathcal{V}^*$.
   - It computes $W := \sum_{i \in [t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$ where $B^* = \mathsf{K}[o]$. Then it samples $V' \in_R \mathbb{Z}_{2^\ell}$ and sets $U' := W - V' \cdot \Delta$, and sends (U', V') to $\mathcal{V}^*$.
   - For the output wire w_h, $\mathcal{S}$ already holds $\mathsf{K}[w_h]$. It samples $\widetilde{\pi} \in_R \mathbb{Z}_{2^\ell}$, and computes $\widetilde{z} := 1 + 2^k \cdot \widetilde{\pi} \in \mathbb{Z}_{2^\ell}$ and $\mathsf{K}[z] := \mathsf{K}[w_h] + 2^k \cdot \mathsf{K}[\pi] \in \mathbb{Z}_{2^\ell}$. Finally it sends $\widetilde{z}$ as well as $\mathsf{M}[z] := \mathsf{K}[z] - \Delta \cdot \widetilde{z} \in \mathbb{Z}_{2^\ell}$ to $\mathcal{V}^*$.

The view of $\mathcal{V}^*$ is distributed exactly as in the real execution of the protocol: The values δ_i and d_i are distributed uniformly in $\mathbb{Z}_{2^\ell}$ and therefore completely mask the circuit inputs and the output values of the multiplication gates, respectively. Moreover, the values U, V computed by the honest $\mathcal{P}$ are also distributed uniformly at random due to the masking with A_0^* and A_1^*, respectively, under the condition that $W = U + V \cdot \Delta$ holds. Hence, the values U', V' sent to $\mathcal{V}^*$ by $\mathcal{S}$ are distributed identically, since we can recover the value W' that an honest $\mathcal{V}$ would compute. Finally, the last message is a valid opening of a commitment [1] where the upper $\ell - k$ bits have been randomized using $[\pi]$. □

Zero-Knowledge Functionality $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$

Prove: On input (Prove, $\{f_1, \ldots, f_t\}$, $\mathbf{w}$) from $\mathcal{P}$ and (Verify, $\{f_1, \ldots, f_t\}$) from $\mathcal{V}$ where $f_1, \ldots, f_n$ are polynomials of degree ≤ 2 in n variables over $\mathbb{Z}_{2^k}$ and $\mathbf{w} \in \mathbb{Z}^n_{2^k}$: Send true to $\mathcal{V}$ iff $f_i(\mathbf{w}) = 0$ for all $i \in [t]$, and false otherwise.

Figure 2.8: Ideal functionality for zero-knowledge proofs for sets of degree-2 polynomials.

General Degree-2 Checks.

Yang et al. [YSWW21a] also provide zero-knowledge proofs for sets of t polynomials of degree d in n variables (in total), where the communication consists of $n+d$ field element – independent of t. With the results proved in Section 2.5.2, we can directly instantiate this protocol with $d = 2$. This allows us to verify arbitrary degree-2 relations including the important use case of inner products. Extending the check for higher-degree relations is principally possible. However, the number of roots of the corresponding polynomials grows exponentially with increasing degree. Hence, to achieve the same soundness, we would need to increase the ring size further, which reduces the efficiency.

We now give the full protocol as well as its security proof.

We formally specify the ideal zero-knowledge functionality for degree-2 relations in Figure 2.8, and prove in the following that protocol $\Pi^k_{\mathsf{QS\text{-}2}}$ (Figure 2.9) realizes this functionality.

Theorem 7. *The protocol $\Pi^k_{\mathsf{QS\text{-}2}}$ (Figure 2.9) securely realizes the functionality $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$ in the $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$-hybrid model when instantiated with the parameters $s := \sigma + \log(\sigma) + 3$ and $\ell := k + 2s$: No unbounded environment $\mathcal{Z}$ can distinguish the real execution of the protocol from a simulated one except with probability $2^{-\sigma+1}$.*

Proof. The proof is similar to the proof of Theorem 6. We divide the proof of security into two parts. First we cover the case of a corrupted prover, then that of a corrupted verifier. In each case we define a PPT simulator $\mathcal{S}$.

Corrupted Prover: The simulation is set up as follows: $\mathcal{S}$ simulates a party $\mathcal{P}^*$ in its head and gives control to $\mathcal{Z}$, and sends ($\mathsf{corrupt}$, $\mathcal{P}$) to $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$. It also simulates an instance of $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. We assume that the circuit $\mathcal{C}$ is known.

1. Simulation of the preprocessing phase: $\mathcal{S}$ simulates the (Init) and (Extend, $n+1$) calls to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. For ($\mathsf{Init}$), it samples $\Delta \in_R \mathbb{Z}_{2^s}$. Since $\mathcal{P}^*$ acts as the sender $\mathcal{S}$ towards $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, it is allowed to choose the sender's output of the (Extend) call. Hence, $\mathcal{S}$ receives $(\widetilde{\mu}_i, \mathsf{M}[\mu_i]), (\widetilde{o}, \mathsf{M}[o]) \in \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$ for $i \in [n]$ from $\mathcal{P}^*$. Then $\mathcal{S}$ can compute matching values $\mathsf{K}[\mu_i], \mathsf{K}[o] \in \mathbb{Z}_{2^\ell}$ according to Equation (2.7).

QuarkSilver for general degree-2 relations $\Pi^k_{\mathsf{QS\text{-}2}}$

The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ have agreed on a set of polynomials in n variabled $f_1, \dots, f_t \in \mathbb{Z}_{2^k}[X_1, \dots, X_n]^{\leq 2}$, and $\mathcal{P}$ holds a witness $\mathbf{w} \in \mathbb{Z}_{2^k}^n$ so that $f_i(\mathbf{w}) = 0$ for all $i \in [t]$. Write $f_i = f_{i,0} + f_{i,1} + f_{i,2}$ where $f_{i,h}$ contains the degree-h terms of f_i.

Preprocessing phase The preprocessing phase is independent of $\mathcal{C}$ and just needs upper bounds on the number of inputs and multiplication gates of $\mathcal{C}$ as input.

1. $\mathcal{P}$ and $\mathcal{V}$ send (Init) to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, and $\mathcal{V}$ receives $\Delta \in \mathbb{Z}_{2^s}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Extend}, \mathrm{n} + 1)$ to $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$, which returns authenticated values $([\mu_i])_{i\in[n]}$ and $[o]$, where all $\widetilde{\mu}_i, \widetilde{o}, \in_R \mathbb{Z}_{2^\ell}$.

Online phase

1. $\mathcal{P}$ commits to its witness $\mathbf{w}$ by sending $\delta_i := w_i - \widetilde{\mu}_i$ for $i \in [n]$ to $\mathcal{V}$, and both parties locally compute $[w_i] := [\mu_i] + \delta_i$.
2. For each f_i, $i \in [t]$:
   - $\mathcal{V}$ computes $B_i := \sum_{h\in[0,2]} f_{i,h}(\mathsf{K}[w_1], \dots, \mathsf{K}[w_n]) \cdot \Delta^{2-h}$
   - $\mathcal{P}$ defines $g_i(X) \in \mathbb{Z}_{2^\ell}[X]$ as $g_i(X) := \sum_{h\in[0,2]} f_{i,h}(\mathsf{M}[w_1] + \widetilde{w}_1 \cdot X, \dots, \mathsf{M}[w_n] + \widetilde{w}_n \cdot X) \cdot X^{2-h}$ and computes coefficients $A_{i,h} \in \mathbb{Z}_{2^\ell}$ such that $g_i(X) = A_{i,0} + A_{i,1} \cdot X + A_{i,2} \cdot X^2$. Note that $A_{i,2} = f_i(w_1, \dots, w_n) = 0$ if $\mathcal{P}$ is honest.
3. $\mathcal{P}$ and $\mathcal{V}$ run the following check:
   (a) Set $A_0^* := \mathsf{M}[o]$, $A_1^* := \widetilde{o}$, and $B^* := \mathsf{K}[o]$ so that $B^* = A_0^* + A_1^* \cdot \Delta$.
   (b) $\mathcal{V}$ samples $\boldsymbol{\chi} \in_R \mathbb{Z}_{2^s}^t$ and sends it to $\mathcal{P}$.
   (c) $\mathcal{P}$ computes $U := \sum_{i\in[t]} \chi_i \cdot A_{0,i} + A_0^* \in \mathbb{Z}_{2^\ell}$ and $V := \sum_{i\in[t]} \chi_i \cdot A_{1,i} + A_1^* \in \mathbb{Z}_{2^\ell}$, and sends (U, V) to $\mathcal{V}$.
   (d) $\mathcal{V}$ computes $W := \sum_{i\in[t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$, and accepts iff $W = U + V \cdot \Delta \pmod{2^\ell}$ holds.

Figure 2.9: Zero-Knowledge protocol for circuit satisfiability in the $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$-hybrid model with $s := \sigma + \log(\sigma) + 3$ and $\ell := k + 2s$ for statistical security parameter σ.

2. To simulate the online phase, $\mathcal{S}$ executes the steps of $\mathcal{V}$ while also keeping track of $\mathcal{P}^*$'s values:

   - For every input $i \in [n]$ it receives $\delta_i \in \mathbb{Z}_{2^\ell}$ from $\mathcal{P}^*$ and computes $\mathcal{V}$'s part of $[w_i] := [\mu_i] + \delta_i$, as well as $\widetilde{w}_i := \widetilde{\mu}_i + \delta_i \in \mathbb{Z}_{2^\ell}$ and $\mathsf{M}[w_i] := \mathsf{M}[\mu_i]$.
   - For the ith polynomial f_i it computes B_i, $A_{i,0}$, and $A_{i,1}$ as in Step 2.
   - $\mathcal{S}$ sends $\boldsymbol{\chi} \in_R \mathbb{Z}_{2^s}^t$ to $\mathcal{P}^*$ and receives two values $U', V' \in \mathbb{Z}_{2^\ell}$ as response. It computes $W := \sum_{i\in[t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$ where $B^* := \mathsf{K}[o]$.
   - If $W \neq U' + V' \cdot \Delta \pmod{2^\ell}$, then $\mathcal{S}$ sends $(\mathsf{Prove}, \{f_1, \dots, f_t\}, \bot)$ on behalf of $\mathcal{P}^*$ to $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$ and simulates an aborting $\mathcal{V}$.
   - Otherwise $\mathcal{S}$ sends $(\mathsf{Prove}, \{f_1, \dots, f_t\}, \mathbf{w})$ with $\mathbf{w} := (\widetilde{w}_i \bmod 2^k)_{i\in[n]}$ on behalf of $\mathcal{P}$ to $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$.

Since $\mathcal{S}$ behaves like an honest $\mathcal{V}$ towards $\mathcal{P}^*$, the view of $\mathcal{P}$ in the simulation perfectly matches its view in the real protocol. Now we need to show that $\mathcal{V}$ outputs the same in the simulation and in the real execution, except with negligible probability. If the honest $\mathcal{V}$ rejects the proof in the protocol, then also $\mathcal{V}$'s output of the ideal $\mathcal{F}^k_{\mathsf{ZK\text{-}2}}$ is false. Now we need to bound the probability that the honest $\mathcal{V}$ accepts the proof, but the ideal $\mathcal{F}^k_{\mathsf{ZK}}$ still outputs false, i.e., $\mathcal{P}^*$ has successfully fooled $\mathcal{V}$ into accepting a proof of a wrong statement.

A corrupted $\mathcal{P}^*$ not knowing a valid witness for the circuit can try to circumvent the check in Step 3: For the ith polynomial f_i, let $0 = f(\widetilde{w}_1, \dots, \widetilde{w}_n) \pmod{2^\ell} + e_i$, where $\widetilde{w}_j \in \mathbb{Z}_{2^\ell}$ is the value contained in the commitment $[w_j]$, for $j \in [n]$, and $e_i \in \mathbb{Z}_{2^\ell}$ is a possible error. Suppose that not all $e_i = 0 \pmod{2^k}$ for $i \in [t]$, i.e., there is an error in the lower k bits that we care about.

Then we have (over $\mathbb{Z}_{2^\ell}$)

$$\begin{aligned}
B_i &= \sum_{h\in[0,2]} f_{i,h}(\mathsf{K}[w_1], \dots, \mathsf{K}[w_n]) \cdot \Delta^{2-h} \\
&= \sum_{h\in[0,2]} f_{i,h}(\mathsf{M}[w_1] + \widetilde{w}_1 \cdot \Delta, \dots, \mathsf{M}[w_n] + \widetilde{w}_n \cdot \Delta) \cdot \Delta^{2-h} \\
&= g_i(\Delta) = A_{i,0} + A_{i,1} \cdot \Delta - e_i \cdot \Delta^2
\end{aligned}$$

where $A_{i,0}$ and $A_{i,1}$ denote the values that an honest $\mathcal{P}$ would compute. So B_i is the result of evaluating a quadratic polynomial at Δ.

By the exact same argument as in the proof of Theorem 6, we can can conclude theat no adversary can break the soundness of the protocol except with probability at most $2^{-\sigma}$.

Corrupted Verifier: The simulation of the verifier's view is straightforward: $\mathcal{S}$ simulates a party $\mathcal{V}^*$ in its head and gives control to $\mathcal{Z}$, and sends $(\mathsf{corrupt}, \mathcal{V})$ to $\mathcal{F}^k_{\mathsf{ZK}}$. It also simulates an instance of $\mathcal{F}^{\ell,s}_{\mathsf{vole2k}}$. We assume that the circuit $\mathcal{C}$ to prove is known.

1. Simulation of the preprocessing phase: $\mathcal{S}$ simulates the (Init) and (Extend, $n+1$) calls to $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$. Since $\mathcal{V}^*$ acts as the sender $\mathcal{S}$ towards $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$, it is allowed to choose its outputs. For (Init), $\mathcal{S}$ receives $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{V}^*$, and $\mathsf{K}[\mu_i], \mathsf{K}[o] \in \mathbb{Z}_{2^\ell}$ for $i \in [n]$ from $\mathcal{V}^*$.

2. To simulate the online phase, $\mathcal{S}$ proceeds as follows:

   - For every input $i \in [n]$ it sends a random $\delta_i \in_R \mathbb{Z}_{2^\ell}$ to $\mathcal{V}^*$, and computes $\mathsf{K}[w_i] := \mathsf{K}[\mu_i] - \delta_i \cdot \Delta$.
   - For the ith polynomial f_i, it computes B_i as the honest $\mathcal{V}$ would do in Step 2.
   - $\mathcal{S}$ receives $\boldsymbol{\chi} \in \mathbb{Z}_{2^s}^t$ from $\mathcal{V}^*$.
   - It computes $W := \sum_{i\in[t]} \chi_i \cdot B_i + B^* \in \mathbb{Z}_{2^\ell}$ where $B^* = \mathsf{K}[o]$. Then it samples $V' \in_R \mathbb{Z}_{2^\ell}$ and sets $U' := W - V' \cdot \Delta$, and sends (U', V') to $\mathcal{V}^*$.

The view of $\mathcal{V}^*$ is distributed exactly as in the real execution of the protocol: The values δ_i are distributed uniformly in $\mathbb{Z}_{2^\ell}$ and therefore completely mask the inputs. Moreover, the values U, V computed by the honest $\mathcal{P}$ are also distributed uniformly at random due to the masking with A_0^* and A_1^*, respectively, under the condition that $W = U + V \cdot \Delta$ holds. Hence, the values U', V' sent to $\mathcal{V}^*$ by $\mathcal{S}$ are distributed identically, since we can recover the value W' that an honest $\mathcal{V}$ would compute. □

2.6 Experiments

In this section we report on the performance of our VOLE protocol $\Pi_{\mathsf{vole2k}}^{r,s}$ (Section 2.4) and our zero-knowledge proof system QuarkSilver (Section 2.5). We implemented the protocols in the Rust programming language using the *swanky* framework[6]. Our implementation is open source and available on GitHub under https://github.com/AarhusCrypto/Mozzarella.

Our implementation is generic, it allows to plugin any ring type that implements certain interfaces. We implement $\mathbb{Z}_{2^\ell}$ based on 64, 128, 192 and 256 bit integers. Depending on the size of ℓ, we choose the smallest of these types. Hence, running the protocol with, e.g., $\ell = 129$ and $\ell = 192$ has exactly the same computational and communication costs. In our experiments, we choose one representative ring for each considered size. It is possible to further optimize the communication cost of the implementation by transmitting exactly ℓ bits instead of the complete underlying integer value at the additional cost for the (un)packing operations.

2.6.1 Benchmarking Environment

All benchmarks were run on two servers with Intel Core i9-7960X processors that have 16 cores and 32 threads. Each server has 128 GiB memory available. They are connected via 10 Gigabit Ethernet with an average RTT of 0.25 ms.

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

We consider different network settings: For the *LAN* setting, we use the network as described above without further restrictions. To emulate a *WAN* setting, we configure Traffic Control in the Linux kernel via the `tc (8)` tool to artificially restrict the bandwidth to 100 Mbit/s, and increase the RTT to 100 ms. Finally, to explore the bandwidth dependence of our VOLE protocol, we consider a set of network settings with 20, 50, 100 and 500 Mbit/s as well as 1 and 10 Gbit/s bandwidth, and an RTT of 1 ms.

2.6.2 VOLE Experiments

In this section, we evaluate the performance of our VOLE protocol $\Pi_{\mathsf{vole2k}}^{\ell,s}$ (Section 2.4). We consider the setting of batch-wise VOLE extension: Given set of n_b base VOLEs, we use our protocols to expand them to $n_o + n_b$ VOLEs to obtain a batch of n_o VOLEs plus n_b VOLEs that can be used as base VOLEs to generate the next batch. We do not consider here how the initial set of base VOLEs are created. As performance measure we use the run-time and communication per generated VOLE correlation in one iteration of the protocol.

LPN Parameter Selection.

For a triple of LPN parameters (m, t, n), our protocol extends $n_b = m + 2 \cdot t$ base VOLEs to n new ones. Hence, for a target batch size n_o, we need to find (m, t, n) such that $n \geq n_o + n_b$ and the corresponding LPN problem is still considered infeasible w.r.t. the security parameters.

As suggested in prior work [SGRR19, YWL+20, WYKW21a], we pick the public LPN matrix $\mathbf{A} \in \mathbb{Z}_{2^\ell}^{m \times n}$ as a generator of a 10-local linear code (i.e. each column of $\mathbf{A}$ contains exactly 10 uniform non-zero entries). As discussed in Section 2.2.5, each non-zero entry is picked randomly from $\mathbb{Z}_{2^\ell}^*$ (i.e. odd), to ensure that reduction modulo 2 does not reduce sparsity. This results in fast computation of the expansion $\mathbf{u} \cdot \mathbf{A}$ (for some $\mathbf{u} \in \mathbb{Z}_{2^\ell}$), as each entry involves only 10 positions of $\mathbf{u}$. We then pick (m, t, n) such that all known attacks on the LPN problem require at least 2^κ operations [BCG+19a, WYKW21a] (see also Section 2.2.5). Note that, as our variant of the regular LPN assumption (Definition 6) leaks blocks of the noise vector, we must pick t such that our protocols are secure in advent of leaking up to $\sigma \in \{40, 80\}$ blocks. To do this, we assume that leaking the noisy index within a single block of $\Pi_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$ directly gives an index of the secret and then subtract the leaked block from the noise vector as well as the corresponding index from the secret and make sure that the new problem is still infeasible to solve.

For a given n_o we experimentally find the LPN parameter set (m, t, n) that gives us the best performance while satisfying the above conditions. Concretely, we start by selecting a number n_o of VOLEs that we want to produce in each iteration. Then we search for parameters (m, t, n) for the leaky LPN problem that gives us $\kappa = 128$ bits of security (see above) such that additionally $n \geq n_o + (m + 2t)$ holds. The latter allows us to generate n_o usable VOLEs in each iteration while keeping $m + 2t$ values to run the next execution of our extension protocol.

In our experiments, we set $n_o \in \{10^7, 10^8\}$. We then tried different secret sizes m in the interval $[10^5, 10^6]$, computed the required number of noise coordinates t so that the leaky LPN problem is sufficiently hard, and benchmarked our protocol. This gave us parameter sets with the same security properties and output size, out of which we selected the best performing one.

If we fix some values of m and n_o (in the ranges stated above) and compare the required noise coordinates with and without the extra leakage, then we need about $1.5\times$ (resp. $2.1\times$) more noise coordinates to compensate for the leakage for $\sigma = 40$ bits (resp. 80 bits) of statistical security.

We chose LPN parameters targeting a level of $\kappa = 128$ bits of computational security, and used the approach of Boyle et al. [BCGI18] to estimate the hardness of the LPN problem. Recently, Liu et al. [LWYY22] noted that this significantly underestimates the hardness of the LPN problem. Using their estimation, our parameters yield about 153–158 bits of security. Hence, we could reduce the parameters to get a more efficient instantiation of our protocol. We chose to use LPN with odd noise values in $\mathbb{Z}_{2^k}$ to resist the reduction attack of Liu et al. [LWYY22], which otherwise reduces the effective noise rate by half. In case of a potential future attack on LPN with odd noise, with the same impact, we would still achieve 103–109 bits of security.

For more details regarding the choice of LPN parameters and how we estimate the hardness of the leaky LPN problem, we refer to Section 2.4.1.

General Benchmarks.

For each statistical security level $\sigma \in \{40, 80\}$, we selected two LPN parameter sets (m, t, n) targeting VOLE batch sizes of $n_o \in \{10^7, 10^8\}$. We execute the protocol in two different network settings with four different ring sizes $\ell \in \{64, 104, 144, 244\}$ (one representative for each of the underlying integer types) for each of the parameter sets. Table 2.1 contains the results of our experiments.

With increasing ring size ℓ the costs increase as the arithmetic becomes more costly and more data needs to be transferred. Moreover, with a larger batch size the costs per VOLE decrease. In terms of run-time and communication costs, it is more efficient to generate a larger amount of VOLEs at once. However, the required resources, e.g., memory consumption, also increase with the batch size. In the WAN setting, a larger batch size is especially more efficient, since the effect of the higher latency is less pronounced on the amortized run-times.

Although the chosen LPN parameter sets worked well in our case, other combinations of m and t can yield a similar performance with same security, while influencing the computation and communication cost slightly. Such an effect can be noticed in the first parameter sets, where the communication cost decreases when going from $\sigma = 40$ to $\sigma = 80$. It is a trade-off, and we deem experimental verification necessary to choose the best-performing parameter set.

Table 2.1: Benchmark results of our VOLE protocol. We measure the run-time of the Extend operation in ns per VOLE and the communication cost in bit per VOLE. The benchmarks are parametrized by the ring size ℓ (i.e., using $\mathbb{Z}_{2^\ell}$). The computational security parameter is set to $\kappa = 128$. For statistical security $\sigma \in \{40, 80\}$, we target batch sizes of $n_o = 10^7$ and $n_o = 10^8$, and use the stated LPN parameters (m, t, n).

σ	ℓ	Run-time		Communication		
		LAN	WAN	$\mathcal{S} \to \mathcal{R}$	$\mathcal{R} \to \mathcal{S}$	total
40	$m = 553\,600$, $t = 2\,186$, $n = 10\,558\,380$					
	64	27.3	190.8	0.467	0.927	1.394
	104	40.7	186.7	0.509	0.955	1.464
	144	55.2	212.6	0.551	0.983	1.534
	244	80.7	255.0	0.593	1.011	1.604
	$m = 773\,200$, $t = 15\,045$, $n = 100\,816\,545$					
	64	20.1	46.0	0.318	0.636	0.954
	104	33.2	58.9	0.347	0.655	1.002
	144	46.7	75.1	0.376	0.674	1.050
	244	76.7	102.8	0.405	0.694	1.098
80	$m = 830\,800$, $t = 2\,013$, $n = 10\,835\,979$					
	64	27.6	171.9	0.431	0.853	1.284
	104	42.6	194.1	0.469	0.879	1.349
	144	59.4	217.1	0.508	0.905	1.413
	244	89.3	277.4	0.547	0.931	1.477
	$m = 866\,800$, $t = 18\,114$, $n = 100\,913\,094$					
	64	21.4	48.2	0.383	0.765	1.148
	104	34.3	61.0	0.418	0.789	1.206
	144	49.2	76.0	0.453	0.812	1.264
	244	79.8	106.8	0.487	0.835	1.322

Table 2.2: Run-times in ns per VOLE in different bandwidth settings, when generating ca. 10^7 VOLEs with 5 threads and statistical security $\sigma \geq 40$. The parameter ℓ denotes the size of a ring or field element. The numbers for Wolverine are taken from [WYKW21a].

	ℓ	20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s	10 Gbit/s
this work	64	110.0	68.7	55.0	50.2	50.6	50.4
	104	142.0	95.2	80.1	73.2	71.5	73.6
	144	178.6	134.7	119.3	111.6	112.6	113.3
	244	266.3	219.1	201.7	194.5	193.7	196.5
Wolverine	61	101.0	87.0	85.0	85.0	85.0	—

Comparison with Wolverine.

We compare the efficiency of our VOLE extension protocol with that of Wolverine [WYKW21a]. While we use different hardware, we try to replicate their benchmarking setup by restricting our benchmark to maximal 5 threads and up to 64 GiB memory, and select LPN parameters to generate $n_o \approx 10^7$ VOLEs. The results are given in Table 2.2, where we list our run-times in different bandwidth settings with the corresponding numbers given in [WYKW21a]. Note that Wolverine uses the prime field $\mathbb{F}_{2^{61}-1}$, whereas we instantiate our protocol with different larger rings $\mathbb{Z}_{2^\ell}$. In network settings with at least 50 Mbit/s bandwidth, we achieve similar or better performance for the ring sizes up to 128 bit.

Bandwidth Dependence.

Table 2.2 also shows how the available bandwidth affects the performance of our protocol. We observe that increasing the network bandwidth beyond 100 Mbit/s does not improve the run-time significantly. This indicates that the required computation is the bottleneck above this point.

2.6.3 Zero-Knowledge Experiments

We explore at what rate our QuarkSilver protocol (Section 2.5) is able to verify the correctness of multiplications. In our experiments we check for $N \approx 10^7$ triples of the form $([w_{i,\alpha}], [w_{i,\beta}], [w_{i,\gamma}])$ for $i \in [N]$ that $w_{i,\alpha} \cdot w_{i,\beta} = w_{i,\gamma} \pmod{2^k}$ holds. Assuming the prover has already committed to $2N$ values $([w_{i,\alpha}], [w_{i,\beta}])$, we execute the following three steps:

1. vole: Perform the Extend operation of $\Pi^{s,\ell}_{\mathsf{vole2k}}$ to create the necessary amount of VOLEs (at least $N + 1$).
2. mult: Step 2 of Π^k_{QS} (Figure 2.7) to commit to the results $w_{i,\gamma} := w_{i,\alpha} \cdot w_{i,\beta}$ of the multiplications.

Table 2.3: Overview of the required ring size ℓ and size s of Δ that different zero-knowledge proofs require to verify circuits over $\mathbb{Z}_{2^k}$ with σ bits of statistical security.

σ	k	$\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ [BBMH+21b]		$\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ [BBMH+21b]		QuarkSilver (**this work**)	
		$s := \sigma$	$\ell := k + s$	$s := \sigma$	$\ell := k + 2s$	$s := \sigma + \log(\sigma) + 3$	$\ell := k + 2s$
40	32	40	72	40	112	49	130
	64	40	104	40	144	49	162
80	32	80	112	80	192	90	212
	64	80	144	80	224	90	244

3. check: Steps 3 and 4 of Π^k_{QS} to verify that the multiplications are correct modulo 2^k.

While the execution of $\Pi^{s,\ell}_{\mathsf{vole2k}}$ in Step 1 is parallelized, the further steps are executed in a single thread, and there is still room for optimizations, e.g., using smaller integers for the coefficients of the random linear combination and better interleaving computation and communication.

For the protocol to be secure, bounds on the ring size ℓ as well as the statistical security σ must be satisfied. We compare those bounds to those of related works in Table 2.3.

For statistical security levels of $\sigma = 40$ and $\sigma = 80$, we run the protocol with ring sizes $\ell = 162$ and $\ell = 244$, respectively. This corresponds to the required ring size ℓ to enable zero-knowledge proof over $\mathbb{Z}_{2^k}$ with $k = 64$. It also covers the $k = 32$ setting, since the corresponding rings (with $\ell \in \{130, 212\}$) are implemented in the same way.

In Table 2.4 we list the achieved run-times and communication costs per multiplication and show how they are distributed over the three steps of the protocol. We clearly see that the costs are dominated by Step 2, where the majority of the communication happens (one $\mathbb{Z}_{2^\ell}$ element per multiplication). Additional benchmarks show that increasing the bandwidth to more than 500 Mbit/s does not increase the performance.

With a completely single-threaded implementation (including single-threaded VOLEs), we can verify about 0.9 million multiplications per second for statistical security parameter $\sigma = 40$ and ring $\mathbb{Z}_{2^{162}}$, compared to (single-threaded) QuickSilver's up to 4.8 million multiplications per second over the field $\mathbb{F}_{2^{61}-1}$, as reported by Yang et al. [YSWW21a]. This is a factor 5.3 difference.

When looking at the performance of $\mathbb{Z}_{2^{162}}$ compared to $\mathbb{F}_{2^{61}-1}$, we see that $\mathbb{Z}_{2^{162}}$ ring elements are represented by three 64 bit integers compared to $\mathbb{F}_{2^{61}-1}$ field elements which fit into a single integer. While this results in 3× more communication, the computational costs are also higher: In microbenchmarks, arithmetic operations in $\mathbb{Z}_{2^{162}}$ are $2.1 - 2.5\times$ slower compared to the corresponding operations in $\mathbb{F}_{2^{61}-1}$ (e.g., $\mathbb{Z}_{2^{162}}$ multiplications require 6 IMUL/MULX instructions, $\mathbb{F}_{2^{61}-1}$ multiplications need one MULX instruction). Moreover, the compiler can automatically vectorize element-wise computations on vectors of field elements with AVX instruction due to the smaller element size, but this is (at least currently) not possible with the larger ring. Computation on rings also results in a slightly higher rate of cache misses, which we attribute to the fact that

Table 2.4: Benchmark results of our QuarkSilver protocol. We measure the run-time of a batch of $\approx 10^7$ multiplications and their verification in ns per multiplication and the communication cost in bit per multiplication. The benchmarks are parametrized by the statistical security parameter σ, and the computational security parameter is set to $\kappa = 128$. For $\sigma = 40$, we use the ring of size $\ell = 162$, for $\sigma = 80$, we use $\ell = 244$.

σ		Run-time		Communication		
		LAN	WAN	$\mathcal{S} \to \mathcal{R}$	$\mathcal{R} \to \mathcal{S}$	total
40	vole	78.5	265.5	0.5	1.0	1.5
	mult	663.2	2 101.5	192.0	0.0	192.0
	check	28.2	38.2	0.0	0.0	0.0
	total	769.9	2 405.2	192.5	1.0	193.5
80	vole	125.3	345.6	0.5	0.9	1.5
	mult	680.7	2 767.2	256.0	0.0	256.0
	check	42.3	52.4	0.0	0.0	0.0
	total	848.3	3 165.2	256.5	0.9	257.5

more field elements than ring elements fit in a cache line, simply due to their size.

We want to stress that this direct comparison is not necessarily fair, though: The Mersenne prime modulus $p = 2^{61} - 1$ has been chosen because it allows to implement the field arithmetic very efficiently. The plaintext space has roughly the same size in both settings (64 vs. 61 bit), but the arithmetic on the secrets is entirely different which is the main difference of our work to the field-based approach of QuickSilver. While QuarkSilver supports 64 bit arithmetic natively (which is one of the main points of considering $\mathbb{Z}_{2^k}$ protocols), things are more complicated with fields. To emulate 64 bit arithmetic in a prime field, the prime modulus has to have size ≥ 128 bit (so no modular wraparound occurs during multiplications) which means more communication and more complicated arithmetic. Then, one also has to commit to the correct reduction modulo 2^{64} and prove that the reduction is computed correctly, e.g., with range proofs or using the truncation protocols of Baum et al. [BBMH+21b] – both are not cheap, in particular given they are needed for each multiplication mod 2^{64} (and possibly additions, too). Moreover, with a prime modulus of this size one cannot take advantage of a Mersenne prime (the nearest Mersenne primes would be $p = 2^{127} - 1$ (too small) and $p = 2^{521} - 1$ (much larger)) to increase computational efficiency.

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), the Independent Research Fund Denmark (DFF) un-

der project number 0165-00107B (C3PO), the Aarhus University Research Foundation, 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). We thank the ENCRYPTO group at TU Darmstadt for allowing us to use their servers for our experiments.

Chapter 3

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

3.1 Introduction

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

A recent line of work has been investigating the scalability of ZK protocols for very large statements, represented as, for instance, circuits with billions of gates. This can be seen in work such as zero-knowledge from garbled circuits [JKO13, FNO15, ZRE15, HK20b] and vector oblivious linear evaluation (VOLE) [WYKW21b, BMRS21b, DIO21b, YSWW21b]. 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.

3.1.1 Our Contributions

In this work, we address the above shortcomings, by introducing efficient conversion protocols for "commit-and-prove"-type ZK, such as recent VOLE-based protocols. We then build on these conversions by presenting new, high-level gadgets for common operations

like truncation and comparison. Finally, we supplement this with efficient ZK protocols for arithmetic circuits over $\mathbb{Z}_{2^k}$, which are also compatible with our previous protocols.

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

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

Note that our protocols are completely agnostic as to the commitment scheme that is used, provided it is linearly homomorphic. However, in practice a fast instantiation can be obtained using information-theoretic MACs based on recent advances in VOLE [BCGI18, SGRR19, BCG+19a, YWL+20] from the LPN assumption. This has been the approach taken in recent VOLE-based ZK protocols [WYKW21b, BMRS21b], which exploit the high computational efficiency and low communication overhead of LPN-based VOLE.

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

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

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

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

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

In our protocol, we introduce several optimizations of this approach, tailored to the ZK setting. Firstly, we observe that in the ZK setting, it is not necessary to create *random* verified `edaBits`, if we can instead just apply the `edaBit` verification protocol

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

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

As a building block of our protocols, we make use of the fact that our `edaBit` consistency check can easily be used to prove that a committed value $x \in \mathbb{Z}_M$ is at most m bits in length, for some public m. We then show that integer truncation in $\mathbb{Z}_M$ can be decomposed into just two length checks, by exploiting the fact that the prover can commit to arbitrary values dependent on the witness. Then, given truncation, we can easily obtain a comparison check, which shows that a committed bit $[b]_2$ encodes $b = (x \stackrel{?}{<} y)$, 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$\mathbb{Z}_{2^k}$ [CDE+18]. Given the MACs, which serve as homomorphic commitments in $\mathbb{Z}_{2^k}$, we show how to efficiently verify multiplications on committed values. We present two possible approaches: the first is based on a simple cut-and-choose procedure, adapted from [WYKW21b] for binary circuits; in the second approach, we adapt the field-based multiplication check from [BMRS21b] to work over rings, which requires some non-trivial modifications.

Since these protocols use VOLE-based information-theoretic MACs, we obtain ZK protocols in the preprocessing model, assuming a trusted setup to distribute VOLE (or short seeds which expand to VOLE [BCGI18]). Removing the trusted setup can be done with an actively secure VOLE protocol over $\mathbb{Z}_{2^k}$. We note that the LPN-based construction of [BCGI18] also works over $\mathbb{Z}_{2^k}$ (as implemented in [SGRR19]), although currently only with passive security. It is an interesting future direction to extend efficient actively secure protocols [BCG+19a, WYKW21b] 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` [EGK+20] 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 [WYKW21b] (9.3 ms amortized per conversion) and 7.5 s using [YSWW21b] (7.3 ms amortized). For 2^{20} 32-bit elements, this increases to 181.7 s for [WYKW21b] (173 ţs amortized) and 92.1 s for [YSWW21b] (87.8 ţs 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 [WYKW21b, BMRS21b], which need to transfer 2–3 field elements .

3.1.2 Related Work

Our work builds upon concretely efficient zero-knowledge protocols from VOLE, which were first given in [WYKW21b, BMRS21b, DIO21b]. While [WYKW21b, BMRS21b] use VOLE-based information-theoretic MACs in a black-box way, Line-Point ZK [DIO21b] takes a non-black-box approach, which reduces communication to just 1 field element per multiplication in a large field. More recently, Quicksilver [YSWW21b] 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 3.6. This is very similar to the approach taken in the concurrent work Mystique [WYX+21b], 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 [BMRS21b], 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 [MRVW21], which provided improved protocols for secure comparison and truncation based on `edaBits`, in the MPC setting. Similarly to our work in ZK, Rabbit allows to avoid the large "gap" between the field size and the desired message space when running these protocols; however, our techniques in the ZK

setting are different.

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

3.2 Preliminaries

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

3.2.1 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 3.2.3 and 3.2.4) 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 $\{1, \ldots, n\}$.

3.2.2 Zero-Knowledge Proofs

Zero-knowledge proofs (of knowledge) are interactive two-party protocols that allow the prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a certain statement is true (and that it possesses a witness to this fact). This happens in a way such that $\mathcal{V}$ does not learn anything else besides this fact that it could not compute by itself. Instead of using the classical definition by Goldwasser et al. [GMR85], we define zero-knowledge using an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ for satisfiability of circuits $\mathcal{C}$: On input $(\mathsf{Prove}, \mathcal{C}, w)$ from $\mathcal{P}$ and $(\mathsf{Prove}, \mathcal{C})$ from $\mathcal{V}$ the functionality $\mathcal{F}_{\mathsf{ZK}}$ outputs $\top$ to $\mathcal{V}$ iff. $\mathcal{C}(w) = 1$ holds, and sends $\bot$ otherwise [WYKW21b].

Following previous works (e.g. [WYKW21b, BMRS21b]), we use the commit-and-prove strategy to instantiate $\mathcal{F}_{\mathsf{ZK}}$ using homomorphic commitments (see Section 3.2.4). These allow the prover $\mathcal{P}$ to commit to its witness w. Then the circuit $\mathcal{C}$ can be evaluated on the committed witness to obtain a commitment to the output, which is opened to prove that indeed $\mathcal{C}(w) = 1$ holds.

3.2.3 VOLE and Linearly Homomorphic MACs

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

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

Oblivious linear-function evaluation (OLE) [NP99, IPS09] is an arithmetic generalization of COT allowing a receiver to evaluate a secret linear equation $\alpha \cdot X + \beta$ (over a field $\mathbb{F}_p$ or ring $\mathbb{Z}_{2^k}$) held by the sender at a point of its choice x to obtain $y = \alpha \cdot x + \beta$. This can be extended into *vector* OLE (VOLE) [ADI+17] where x and β are vectors of the same length rather than single field elements. *Subfield* VOLEs [BCG+19b] 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 3.5.2 where we adapt the work of [CDE+18] to the zero-knowledge setting. For fields $\mathbb{Z}_p$, we use BeDOZa-style MACs [BDOZ11] which can be generated as follows: To authenticate values $x_1, \dots, x_n \in \mathbb{Z}_p$ known to $\mathcal{P}$, random keys $\Delta, \mathsf{K}[x_1], \dots, \mathsf{K}[x_n] \in_R \mathbb{Z}_p$ are chosen by $\mathcal{V}$, and then $\mathcal{P}$ obtains the MACs $\mathsf{M}[x_i] \leftarrow \Delta \cdot x_i + \mathsf{K}[x_i] \in \mathbb{Z}_p$. We use the notation $[x_i]_p$ for this. To open $[x]_p$, $\mathcal{P}$ sends x and $\mathsf{M}[x]$ to $\mathcal{V}$, who checks that $\mathsf{M}[x] = \Delta \cdot x + \mathsf{K}[x]$ 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 $\mathsf{M}[z] := a \cdot \mathsf{M}[x] + \mathsf{M}[y]$ and $\mathsf{K}[z] := a \cdot \mathsf{K}[x] + \mathsf{K}[y] - \Delta \cdot b$. 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 $\mathsf{K}[x_i]$ 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 [WYKW21b, BMRS21b].

3.2.4 Homomorphic Commitment Functionality

As discussed in Section 3.2.2, 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 3.2.3). For finite fields $\mathbb{Z}_p$, this was shown with the protocols Wolverine [WYKW21b] and Mac'n'Cheese [BMRS21b]. We refer to their works for details. For rings $\mathbb{Z}_{2^k}$, we present an instantiation in Section 3.5.2.

We formally define the homomorphic commitments using the ideal functionality $\mathcal{F}^R_{\mathsf{ComZK}}$ given in Figure 3.1. The parameter R denotes the message space, which is in our case either a ring $\mathbb{Z}_{2^k}$ or a field $\mathbb{Z}_p$. In addition to the common Input and Open operations, which enables $\mathcal{P}$ to commit to a value and reveal it to $\mathcal{V}$ at a later point, we also model Random and CheckZero, for generating commitments of random values and verifying that a committed value equals zero, respectively, which enables more efficient

implementations. Moreover, $\mathcal{F}^R_{\mathsf{ComZK}}$ allows via Affine to compute affine combinations of committed values with public coefficients yielding again a commitment of the result. Finally, CheckMult allows to verify that a set of 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.

3.2.5 Extended Doubly-Authenticated Bits

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

Their generalization, called `edaBits` (due to Escudero et al.[EGK+20]), is defined as m bits $b_0, \dots, b_{m-1}$ which are each authenticated in the binary domain while their sum is authenticated in the arithmetic one, i.e. $([b_0]_2, \dots, [b_{m-1}]_2, [b]_M)$, 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 [EGK+20] consists of two different phases: in the first phase, each party locally samples `edaBits` and proves to all other parties that they were computed correctly. Then, in a second phase, these local contributions are combined to global, secret `edaBits`. In our setting however, only the prover will use `edaBits`, thus it is clear that the second phase can be omitted. Our sampling protocol will only have to ensure that each `edaBit` $([x_0]_2, \dots, [x_{m-1}]_2, [\![x]\!])$ is indeed consistent, i.e. that $x = \sum_{i=0}^{m-1} x_i 2^i \bmod M$.

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

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

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

3. The prover and verifier engage in a check that ensures that the committed values of the prover are consistent. For this, the prover first opens C of the $NB + C$ committed tuples to show consistency (where the choice is made by the verifier).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.1: Functionality modeling homomorphic commitments of values in the ring R.

Then the NB `edaBits` are distributed into N buckets of size B. $B-1$ of the `edaBits` are then used to verify that the remaining `edaBit` per bucket is consistent without leaking information about it.

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

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

3.3 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.

3.3.1 Conversions and `edaBits` in ZK

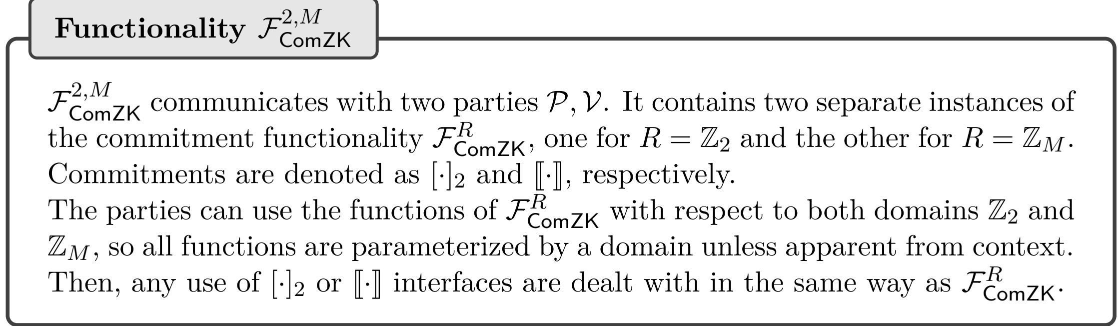
In secure multi-party computation, `edaBits` are used to compute a conversion of a value $[\![x]\!]$ that is secret-shared among multiple parties. In the zero-knowledge setting, the prover knows the underlying value x, so there is no need to convert $[\![x]\!]$ securely into its bit decomposition $([x_0]_2, \ldots, [x_{m-1}]_2)$ 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 7 (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 $([x_0]_2, \ldots, [x_{m-1}]_2, [\![x]\!])$ is called a* conversion tuple. *We call $([x_0]_2, \ldots, [x_{m-1}]_2, [\![x]\!])$ consistent iff $x = \sum_{i=0}^{m-1} 2^i x_i \bmod M$.*

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

Our protocols perform conversions on committed values in $\mathbb{Z}_2$ and $\mathbb{Z}_M$, where we recall that M is either a large prime or 2^k. We model these commitments using the functionality $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ in Figure 3.2, which extends two instances of $\mathcal{F}_{\mathsf{ComZK}}^{R}$ for $R = \mathbb{Z}_2$ and $R = \mathbb{Z}_M$ and simply parses all method calls to the respective instance.

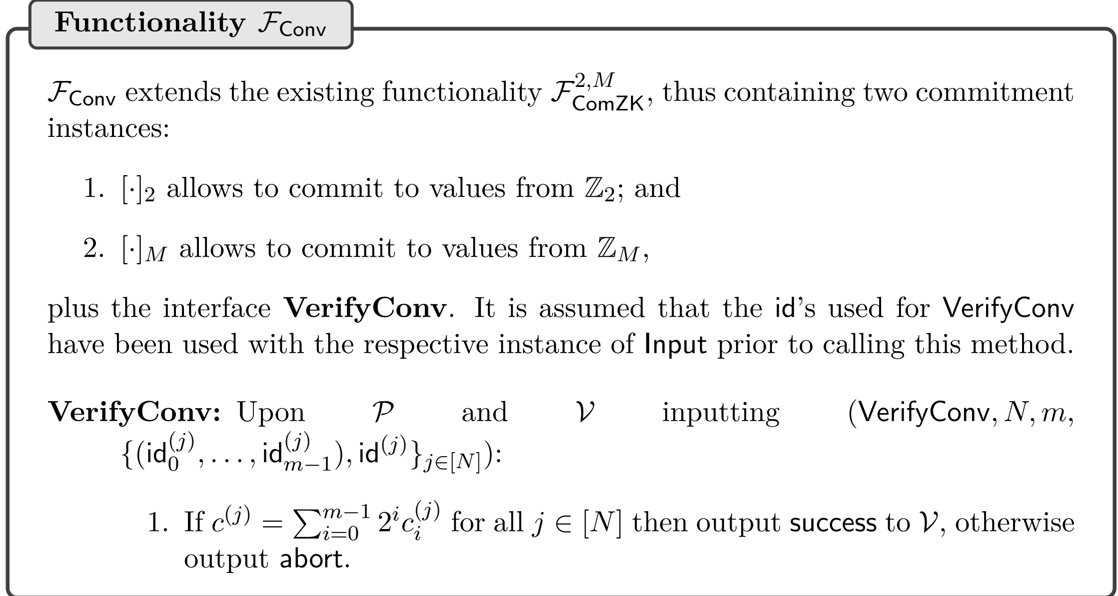
Finally, we define the ideal functionality for verifying conversions $\mathcal{F}_{\mathsf{Conv}}$ in Figure 3.3. This functionality extends $\mathcal{F}_{\mathsf{ComZK}}^{2,M}$ with a single method $\mathsf{VerifyConv}$. It essentially checks whether or not the two representations of some hidden value are consistent.



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

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

Figure 3.2: Ideal functionality modeling communication using commitments over multiple domains.



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

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

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

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

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

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

Figure 3.3: Functionality $\mathcal{F}_{\mathsf{Conv}}$ checking `edaBits`

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

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

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

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

Figure 3.4: Functionality $\mathcal{F}_{\mathsf{Dabit}}$ checking `daBits`.

3.3.2 The Conversion Verification Protocol Π_{Conv}

The following protocol Π_{Conv} verifies the correctness of a batch of N conversion tuples. Π_{Conv} uses $\mathcal{F}_{\mathsf{Dabit}}$ (Figure 3.4) to verify correctness of `daBits` (recall, a `daBit` is an `edaBit` of length 1), which is needed in one stage of the protocol. Later, we show how to remove most of the `daBit` check to improve efficiency.

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

On a high level, Π_{Conv}, in Figure 3.5, consists of three phases:

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

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

3. We place each conversion tuple into one of N buckets, each of which contains a conversion tuple $([c_0]_2, \ldots, [c_{m-1}]_2, [\![c]\!])$, and a set of B `edaBits` $\{([r_0]_2, \ldots, [r_{m-1}]_2, [\![r]\!])_i\}_{i=0}^{B-1}$. 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 $[\![c + r_j]\!] = [\![c]\!] + [\![r_j]\!]$ 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).

Protocol Π_{Conv}

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

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

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

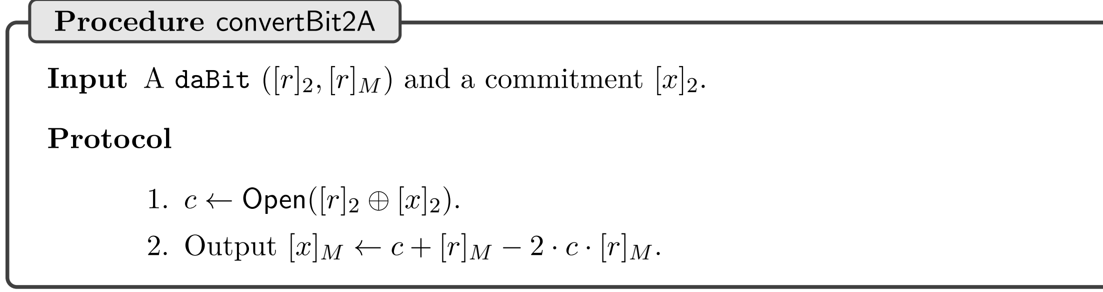
// $\mathcal{P}$ and $\mathcal{V}$ shuffle the auxiliary values and a subset gets opened and verified.

3. $\mathcal{V}$ samples uniformly random permutations $\pi_1 \in S_{NB+C}, \pi_2 \in S_{NB}, \pi_3 \in S_{NBm+Cm}$ and sends them to $\mathcal{P}$.
4. Both parties shuffle the `edaBits` $[r_0^{(j)}]_2, \ldots, [r_{m-1}^{(j)}]_2, [r^{(j)}]_M$ locally according to π_1. They then shuffle $[b_2^{(j)}]_2, [b_M^{(j)}]_M$ according to π_2 and $[x^{(j)}]_2, [y^{(j)}]_2, [z^{(j)}]_2$ according to π_3.
5. Run a cut-and-choose procedure as follows:
   (a) $\mathcal{P}$ opens $\{[r_0^{(j)}]_2, \ldots, [r_{m-1}^{(j)}]_2, [r^{(j)}]_M\}_{j=NB+1}^{NB+C}$ (the last C `edaBits`) towards $\mathcal{V}$, who in turn checks that $r^{(j)} \stackrel{?}{=} \sum_{i=0}^{m-1} 2^i \cdot r_i^{(j)}$.
   (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 $\mathsf{CheckZero}([z^{(j)}]_2 - x^{(j)} \cdot y^{(j)})$ 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 $([e_0]_2, \ldots, [e_m]_2) \leftarrow \mathsf{bitADDcarry}([c_0]_2, \ldots, [c_{m-1}]_2, [r_0]_2, \ldots, [r_{m-1}]_2)$.
   (c) Convert $[e_m]_M \leftarrow \mathsf{convertBit2A}([e_m]_2)$ using the $(i-1) \cdot B + j$'th `daBit` $([b]_2, [b]_M)$.
   (d) Let $[e']_M \leftarrow [c+r]_M - 2^m \cdot [e_m]_M$.
   (e) Let $e_i \leftarrow \mathsf{Open}([e_i]_2)$ for $i = 0, \ldots, m-1$. Then run $\mathsf{CheckZero}([e']_M - \sum_{i=0}^{m-1} 2^i \cdot e_i)$.
7. If any of the checks fail, $\mathcal{V}$ outputs abort. Otherwise it outputs $\mathsf{success}$.

Figure 3.5: Protocol Π_{Conv} to verify Conversion Tuples



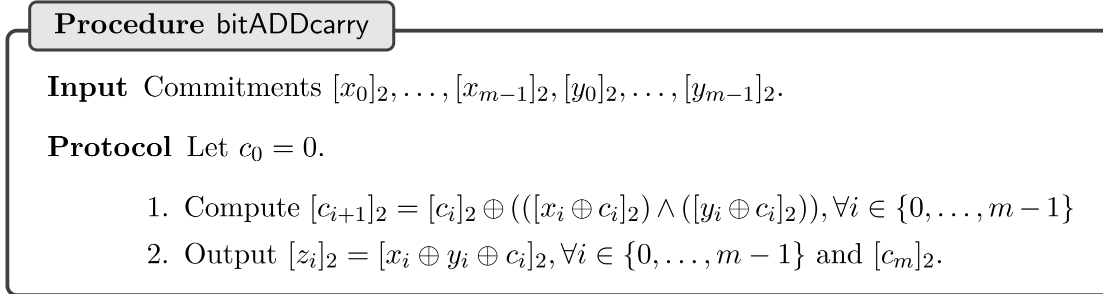
Procedure convertBit2A

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

Protocol

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

Figure 3.6: Procedure to convert bit from $\mathbb{Z}_2$ to $\mathbb{Z}_M$.



Procedure bitADDcarry

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

Protocol Let $c_0 = 0$.

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

Figure 3.7: A ripple-carry adder

For the checks within each bucket, we use the two sub-protocols convertBit2A (Figure 3.6) and bitADDcarry (Figure 3.7). The former converts an authentication of a bit $[b]_2$ into an arithmetic authentication $[\![b]\!]$ while the latter adds two authenticated values $([x_0]_2, \ldots, [x_{m-1}]_2)$ and $([y_0]_2, \ldots, [y_{m-1}]_2)$. This uses a ripple-carry adder circuit, which satisfies the following weak tamper-resilient property, as observed in [EGK+20].

Definition 8. *A binary circuit $C : \mathbb{Z}_2^{2m} \rightarrow \mathbb{Z}_2^{m+1}$ is weakly additively tamper resilient, if given any additively tampered circuit C^*, obtained by flipping the output of any fixed number of AND gates in C, one of the following two properties hold:*

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

Note that the type of additive tampering in Definition 8 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 $[\![c + r_j]\!]$ 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 $[\![e_m]\!]$ to remove the carry from $[\![c + r_j]\!]$ by computing $[\![e']\!] = [\![c + r_j]\!] - 2^m \cdot [\![e_m]\!]$. Now all that remains is to open $[\![e']\!]$ (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 $e' = c + r \mod 2^m$. 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 3.6.

3.3.3 Proof of security

For clarity (due to its sheer size), this proof is written in its own section (Section 3.3.4). We summarize the proof below.

In order to prove the security of Π_{Conv}, we first observe that instead of letting $\mathcal{P}$ choose multiplication triples, we might equivalently model this by letting $\mathcal{P}$ specify circuits instead (that will be evaluated instead of the Ripple Carry Adder). Then, we define an abstraction of the protocol as a balls-and-bins type game, similar to [EGK+20], and analyze the success probability of an adversary in this game.$\mathcal{A}$ winning in this abstraction rather than in the protocol Π_{Conv}. We make this abstraction, as a straightforward analysis of the conversion protocol is rather complex. This is due to there being multiple ways for $\mathcal{A}$ to pass the check with a bad conversion tuple. The first is by corrupting K conversion tuples, then corrupting $K \cdot B$ `edaBits` and hoping that these end up in the right buckets, canceling out the errors in the conversion tuples. The second approach is to corrupt a set of `edaBits` and then guess the arrangement of these, thus yielding how many circuits $\mathcal{A}$ would have to corrupt in order to cancel out the errors of the conversion tuples. Furthermore, conversion tuples (andă`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 8. *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 Section 3.3.4. The approach is similar to that of [EGK+20], however in our case since the conversion tuples are now fixed to be one per bucket, we have not taken a random permutation across all edaBits and conversion tuples. Therefore, we need a different analysis to show that this restriction on the permutation still suffices.

Using this, in Section 3.3.4 we then prove security of Π_{Conv}:

Theorem 9. *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 3.5) UC-realises $\mathcal{F}_{\mathsf{Conv}}$ (Figure 3.3) in the $\mathcal{F}_{\mathsf{Dabit}}$-hybrid model. Specifically, no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most 2^{-s}.*

3.3.4 Proof of Security of the Basic Conversion Protocol

This section contains the proof of security for our conversion tuple verification protocol Π_{Conv} (Section 3.3.2). In Section 3.3.4 we seek to create an abstraction to the cut-and-choose procedure (Step 5 of Π_{Conv}), that is easier to analyse. Firstly we present a game called the **RealGame** which differs only slightly from the cut-and-choose procedure. Secondly we present the **SimpleGame** which further abstracts details away, but we argue that it still represents the cut-and-choose procedure. In Section 3.3.4 we show that our cut-and-choose procedure used within Π_{Conv} is secure, by showing that for specific parameters, a malicious prover is incapable of winning the **SimpleGame** by passing the check with an inconsistent conversion tuple, unless with negligible probability. Lastly, in Section 3.3.4 we prove that 9 holds true, by showing that Π_{Conv} (Figure 3.5) UC-realises $\mathcal{F}_{\mathsf{Conv}}$ (Figure 3.3) in the $\mathcal{F}_{\mathsf{Dabit}}$-hybrid model.

Security of the Cut-and-Choose Game

Initially (In section 3.3.4) we focus on abstracting the cut-and-choose procedure by defining a new game called the **RealGame**. Within this game, some details have been omitted, such as the prover and verifier no longer producing multiplication triples, but instead the prover picks an additively tampered binary circuit directly. In Section 3.3.4 we then further abstract the cut-and-choose game, by defining the **SimpleGame** in which `edaBits`, multiplication triples or `daBits` no longer exist, but instead only balls and triangles. This turns the analysis into a balls-and-bins type game and allows us to show that the prover only hits specific combinations allowing the prover to cheat, with negligible probability.

The RealGame In order to prove the security of Π_{Conv}, we define an abstract game **RealGame** (Figure 3.8). In this abstraction, the prover (or $\mathcal{A}$) will pick additively tampered binary circuits directly, rather than individual multiplication triples. Apart from this change, the check run on each Conversion Tuple within each bucket is the same (step 6). We define all elements used to verify the consistency as *checking tuples*. These checking tuples contain an `edaBit`, a `daBit` and a potentially tampered circuit.

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

The SimpleGame We define an additional abstraction to the **RealGame**. This **SimpleGame** is even simpler in the sense that it no longer considers `edaBits`, triples

The RealGame

1. $\mathcal{A}$ prepares $N + (NB + C)$ authenticated `edaBits` (representing the N conversion tuples and $NB + C$ `edaBits` required to run the check), $\{([r_0^j]_2, \ldots, [r_{m-1}^j]_2, [r^j]_M)\}_{j \in [N+NB+C]}$, $NB + C'$ potentially tampered circuits $\{C^{j,*}\}_{j \in [NB+C']}$ and lastly NB `daBits` $\{([b^j]_2, [b^j]_M)\}_{j \in [NB]}$. These are all send to the challenger.

2. The challenger shuffles the N `edaBits` representing conversion tuples, then shuffles the remaining as well as the circuits using 3 permutations.

3. The challenger opens C checking `edaBits` and C' of the circuits. If any of the `edaBits` or circuits are inconsistent, terminate.

4. The challenger pairs up the remaining NB circuits with the NB `daBits` according to their permutations.

5. The challenger lets the shuffled list of the N `edaBits` be the first `edaBit` of each bucket before pairing up the remaining NB checking `edaBits` and NB circuits with the buckets.

6. Within each of the buckets, for every pair of `edaBits` $([r_0]_2, \ldots, [r_{m-1}]_2, [\![r]\!])$ (where this is the first of the bucket) and $([s_0]_2, \ldots, [s_{m-1}]_2, [\![s]\!])$ take the next circuit C^* and compute $(c_0, \ldots, c_m) \leftarrow C^*(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1})$. Compute $c \leftarrow \sum_{i=0}^{m-1} c_i 2^i$ and check $r + s - 2^m \cdot c_m \stackrel{?}{=} c$.

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

Figure 3.8: An abstraction of the cutNchoose protocol used to verify conversion tuples

The SimpleGame

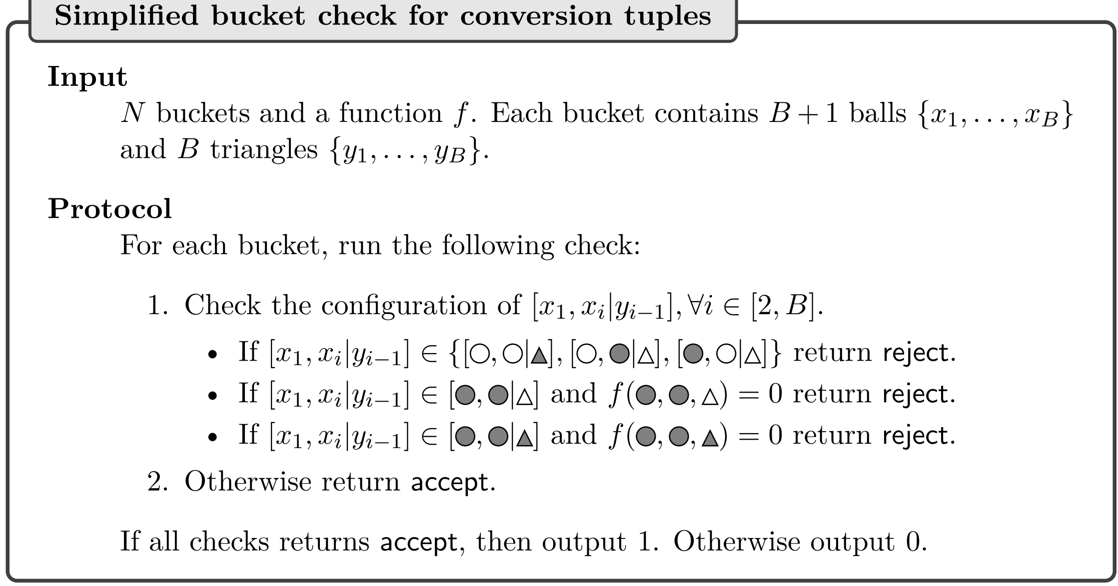
1. $\mathcal{A}$ prepares $N + (NB + C)$ balls and corrupts b of the $NB + C$ balls. These are all sent to the challenger.
2. The challenger opens C of the $NB + C$ balls at random and checks whether all C are good. If any of these balls are bad, then terminate.
3. The challenger shuffles the initial N and the remaining NB balls individually and then associates the initial N balls individually with N buckets. The challenger then randomly assigns the remaining NB balls to the N buckets which each have capacity B. The arrangement of balls is send to $\mathcal{A}$.
4. $\mathcal{A}$ prepares $NB + C'$ triangles and corrupts t of them. These are all sent to the challenger.
5. The challenger opens C' of the triangles at random and checks whether all C' are good. If any of these triangles are bad, then terminate.
6. The challenger shuffles the remaining NB triangles and randomly assigns these to the N buckets.
7. The challenger runs the bucketcheck procedure (Figure 3.10).
8. If bucketcheck returns 1, the challenger accepts the first ball of each bucket, otherwise terminate.

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

Figure 3.9: A game working as a further abstraction of the **RealGame**

or `daBits`. We argue that this simplified game **SimpleGame** models the **RealGame**, before analyzing the probability of success for the **SimpleGame**.

Within the **SimpleGame** `edaBits` are transformed into balls in such a way that a good `edaBit` is a clear ball (○) and bad (corrupt) `edaBits` are shades of gray balls (I.e. ● or ●) where each shade defines a different kind of corruption. Likewise, a good circuit is a clear triangle (△) and bad circuits are gray triangles (▲). A bad ball (or triangle) is bad in the sense that it helps the adversary win the game. Everyone is given access to the public function f that takes two balls and a triangle and outputs 0 or 1. This function f is isomorphic to the winning condition in step 6 of **RealGame** and is modelled by the bucket check procedure shown in Figure 3.10. Finally, the adversary wins if bucketcheck does not abort, meaning that $\mathcal{A}$ passed all the checks, and there is at least one bad ball in the output.



Simplified bucket check for conversion tuples

Input

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

Protocol

For each bucket, run the following check:

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

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

Figure 3.10: A bucket check procedure used to check consistency of conversion tuples in the **SimpleGame**

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

For clarity, we list the different advantageous (for the adversary) combinations of balls and triangles in Table 3.1. If the configurations within the buckets match those of the first three entries of Table 3.1 then bucketcheck will not terminate. If any match the penultimate entry or the final entry, then bucketcheck terminates if the output of f is 0.

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

Lemma 6. *Security against all adversaries in **SimpleGame** implies security against all adversaries in **RealGame**.*

Proof. (Sketch.) We argue the security of our revised **RealGame** by showing that if there exists an efficient adversary $\mathcal{B}$ that wins **RealGame** with non-negligible proba-

Table 3.1: Favorable combinations of balls and triangles for the adversary

Circles		Triangles
○	○	△
○	●	▲
●	○	▲
●	●	△ / ▲
●	⬤	▲ / △

bility, then there exists an efficient adversary $\mathcal{A}$ against **SimpleGame** that wins with non-negligible probability. $\mathcal{A}$ will simulate the **RealGame** challenger and then use $\mathcal{B}$ to win **SimpleGame**.

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

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

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

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

Analysis of the Cut and Choose procedure

We will now prove that this cut-and-choose protocol is sound, as stated in 8.

In order to pass the `bucketcheck`, the adversarial prover will have to fill every bucket with (ball, ball, triangle) arrangements according to Table 3.1.

We will now analyze the probability of success of an adversarial prover, i.e. that the prover gets through all three checks described in `bucketcheck` with at least one inconsistent conversion tuple. Throughout this analysis we will use b to denote the number of bad balls (of the `edaBits` within checking tuples) and t to denote the number

of bad triangles. We assume that $N \geq 2^{\frac{s}{B-1}}$.

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

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

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

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

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

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

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

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

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

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

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

Lemma 8. *The probability of $\mathcal{A}$ passing the second check in **SimpleGame** is less than 2^{-s}, if the adversary corrupts more than β fraction of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens C' triangles.*

The Lemmas 7 & 8 imply that whenever the fraction or bad ball or triangles is large enough, the adversary would already lose during the first two checks. We now analyze the probability of hitting arrangements that pass bucketcheck in such a way that $\mathcal{A}$ wins with respect to small enough fractions of faulty balls and triangles.

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

{(●, ○, ●), (⚫, ○, ○), (○, ○, ○)}

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

Using Table 3.1 we see there are two possible favorable combinations of triangles.

{(△, ▲, △), (▲, △, △), (△, △, △)}
{(▲, ▲, ▲), (▲, △, △), (△, △, △)}

This is due to the last entry of Table 3.1 saying that whenever there are two bad balls, the check may pass using a good triangle ([●, ●, △]) or a bad triangle ([●, ●, ▲]). In this example, let f(●, ●, ▲) $= 1$ (and f(○, ⚫, ▲) $= 1$) in which case the second arrangement is the favorable arrangement.

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

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

where $t = NB\beta$. Thus,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Lastly we note that

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

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

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

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

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

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

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

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

Lastly, we consider $t = 2$. Now the adversary has to compensate for an extra bad triangle, compared to the previous case. In this case we encounter a specific arrangement

that could be an issue regarding our way of analysing these cases. We will argue, however, that this is not a problem.

The following arrangement

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

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

{(△, △, △), (△, △, △), (▲, △, ▲), (△, △, △)}

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

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

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

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

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

$$\mathsf{Pr}[\mathcal{A} \text{ passes } \mathsf{bucketcheck}] \leq \binom{N}{K}\binom{KB}{g_1+b_1}\binom{(N-K)B}{b_2}\frac{1}{\binom{NB}{KB-g_1+b_2}}\frac{1}{\binom{NB}{t}} \tag{3.1}$$

where g_1 is the total number of good balls in the K buckets containing bad conversion tuples, b_1 is the total number bad balls of a different kind from ◍ (represented as ●) and

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

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

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

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

{(◉,◉,◉,◉),(◉,○,○,○),(○,○,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,◉,○,○),(◉,○,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,◉,◉,○),(◉,○,○,○),(○,◉,○,○)}

{(◉,◉,◉,◉),(◉,◉,◉,◉),(◉,○,○,○),(○,◉,◉,○)}

{(◉,◉,◉,◉),(◉,●,○,○),(○,○,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,●,●,○),(○,○,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,●,●,●),(○,○,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,◉,●,○),(○,◉,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,◉,●,●),(○,◉,○,○),(○,○,○,○)}

{(◉,◉,◉,◉),(◉,◉,◉,●),(○,◉,○,○),(◉,○,○,○)}

Using eq. 3.1 we can compute the probabilities of success for all these cases at $K=1$ and $K=N-1$ in order to find the best possible scenario for $\mathcal{A}$. By doing so, we conclude that the 9'th case has the highest probability of success at $K=N-1$. Considering this case, the probability is computed as,

$$\mathsf{Pr}[\mathcal{A}\text{ passes }\mathsf{bucketcheck}]\leq\binom{N}{N-1}\binom{(N-1)\cdot 4}{2}\cdot\binom{4}{1}\frac{1}{\binom{4N}{4(N-1)+1}}\frac{1}{\binom{4N}{3}}$$

$$
\begin{aligned}
&= N \cdot \binom{4N-4}{2} \cdot 4 \cdot \frac{(4N-3)! \cdot 3!)}{(4N)!} \cdot \frac{3! \cdot (4N-3)!}{(4N)!} \\
&= 8(N-1) \cdot (4N-5) \cdot \frac{3}{4N} \cdot \frac{3}{4N} \cdot \frac{2}{4N-1} \cdot \frac{2}{4N-1} \cdot \frac{1}{4N-2} \cdot \frac{1}{4N-2} \\
&\leq 8(N-1) \cdot (4N-5) \cdot (2^{-s/3})^6, \text{ given } N \geq 2^{s/3}
\end{aligned}
$$

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

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

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},\bullet,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},\bullet,\bullet,\circ,\circ),(\circ,\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},\bullet,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},\bullet,\bullet,\bullet,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},\bullet,\bullet,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},\bullet,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},\circ),({\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},\bullet,\bullet,\bullet,\bullet),(\circ,\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},\bullet,\bullet,\bullet),({\color{gray}\bullet},\circ,\circ,\circ,\circ),(\circ,\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},\bullet,\bullet),({\color{gray}\bullet},\circ,\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},\bullet),({\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ,\circ),({\color{gray}\bullet},\circ,\circ,\circ,\circ)\}$$

$$\{({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet},{\color{gray}\bullet}),({\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ,\circ),({\color{gray}\bullet},{\color{gray}\bullet},\circ,\circ,\circ)\}$$

Again, by using eq. 3.1, we compute the different probabilities of success of all 15 cases in similar fashion to our analysis of $B = 3$ and $B = 4$ and conclude that the 13'th case has the highest probability of winning at $K = N - 1$. We now describe this probability.

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

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

We summarize the analysis as follows.

Lemma 9. *The probability of $\mathcal{A}$ passing* bucketcheck *in **SimpleGame** is less than 2^{-s} given $N \geq 2^{s/B-1}$ and the challenger opens $C = B$ balls and $C' = B$ triangles during the first two checks of **SimpleGame** for $B \in \{3, 4, 5\}$ given s such that $\frac{s}{B-1} > B$.*

Proof. This lemma follows from a case-by-case analysis of the bucketcheck procedure, in combination with Lemmas 7 & 8. □

Combining Lemma 6 and Lemma 9 completes the proof of Theorem 8.

Proof of Security of the Protocol Π_{Conv} (Theorem 9)

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

1. In the setup-phase, $\mathcal{P}^*$ sends `edaBits` $\{(r_0^j, \ldots, r_{m-1}^j, r^j)\}_{j \in [NB+C]}$, `daBits` $\{(b^j, b'^{,j})\}_{j \in [NB+s]}$ and triples $\{(x^j, y^j, z^j)\}_{j \in [NBm+Cm]}$. $\mathcal{S}$ records and forwards all of these to $\mathcal{F}_{\mathsf{Conv}}$.

2. $\mathcal{S}$ runs $\mathcal{F}_{\mathsf{Dabit}}$ with the input provided by $\mathcal{P}^*$. If $\mathcal{F}_{\mathsf{Dabit}}$ returns abort, then send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate. Otherwise continue.

3. $\mathcal{S}$ randomly sample permutations $\pi_1 \in S_{NB+C}, \pi_2 \in S_{NB+s}$ and $\pi_3 \in S_{NBm+Cm}$ and send these to $\mathcal{P}^*$ before shuffling the values provided by $\mathcal{P}^*$.

4. The simulator emulates the **Cut**-phase by calling Open on the last C `edaBits` and triples and ensuring the consistency of each. If any check fail, send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate.

5. For each bucket during the **Choose**-phase, $\mathcal{S}$ emulate bitADDcarry by running like an honest verifier and convertBit2A by calling this using $\mathcal{F}_{\mathsf{Conv}}$.

6. $\mathcal{S}$ runs the rest of the protocol as an honest verifier. If the honest verifier outputs abort, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate. If the honest verifier outputs success, then $\mathcal{S}$ sends $(\mathsf{VerifyConv}, N, \{[c_0^{(j)}]_2, \ldots, [c_{m-1}^{(j)}]_2, [c^{(j)}]_M\}_{j \in [N]})$ to $\mathcal{F}_{\mathsf{Conv}}$ (essentially forwarding the call made originally by $\mathcal{P}^*$).

The messages that $\mathcal{P}^*$ receives from $\mathcal{S}$ have the same distribution as in the real protocols. Whenever the verifier simulated by $\mathcal{S}$ outputs abort (as in the protocol), then the verifier in the ideal setting outputs abort as well (since $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Conv}}$). The only case where $\mathcal{P}^*$ may distinguish between the ideal and real, is if the simulated verifier run by $\mathcal{S}$ outputs success but at least one conversion tuple is inconsistent, in which case $\mathcal{F}_{\mathsf{Conv}}$ will abort. But if *at least* one conversion is inconsistent, then by Theorem 8 the

probability with which $\mathcal{P}^*$ avoids being caught in Step 6 of Π_{Conv} is at most 2^{-s}.

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

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

2. If $\mathcal{V}^*$ at any outputs abort, then send abort to $\mathcal{F}_{\mathsf{Conv}}$ and terminate.

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

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

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

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

7. Lastly, $\mathcal{S}$ sends

$$(\mathsf{VerifyConv}, N, \{[c_0^{(j)}]_2, \ldots, [c_{m-1}^{(j)}]_2, [c^{(j)}]_M\}_{j\in[N]})$$

to $\mathcal{F}_{\mathsf{Conv}}$ (again, forwarding the initial call) and outputs whatever $\mathcal{V}$ outputs.

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

□

3.3.5 Faulty daBits

A crucial error was discovered in this section, which forces a bound on the bit-lengths of the values that we can check with the conversion check, when using the strategy involving faulty daBits. We consider this bound too unrealistic for the use-cases we present, and as a result, this section has been removed for now.

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

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

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

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

Figure 3.11: Functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$ that verifies a truncation

3.4 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 $\mathbb{Z}_M$.

3.4.1 Truncation

In Figure 3.11 we present a functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$ that takes a batch of commitments $[\![a_j]\!]$ and their supposed truncations (by m^j bits) $[\![a'_j]\!]$. The functionality ensures that the truncations are correct, namely, $a'_j = \lfloor \frac{a_j}{2^{m^j}} \rfloor$. Note that this functionality we realise is flexible, in that it can support a large batch of truncations, each of which may be of a different length.

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

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

Having access to $[\![a_{tr}]\!]$ and $[\![a']\!]$ 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 $[\![a_{tr}]\!]$ and $([a^0_{tr}]_2, \dots, [a^{l-m-1}_{tr}]_2)$ not only shows consistency between the binary and arithmetic representations, but also that $[\![a_{tr}]\!]$ can be represented by $l - m$ or less bits (same goes for $[\![a']\!]$ and its bit decomposition).

We first define an ideal functionality $\mathcal{F}_{\mathsf{CheckLength}}$ (Figure 3.12) that encapsulates this concept of using $\mathcal{F}_{\mathsf{Conv}}$ as a way of bounding the size of an authenticated value.

The protocol $\Pi_{\mathsf{CheckLength}}$ ensures that $[\![a]\!]$ can be represented by m bits, as it proves consistency between the two representations of a. The security of this protocol directly follows from using $\mathcal{F}_{\mathsf{Conv}}$. The cost of the protocol also directly follows from the consistency check described in Figure 3.5.

Note that $\Pi_{\mathsf{CheckLength}}$ and $\Pi_{\mathsf{VerifyTrunc}}$ do not utilise anything specific about M except $2^l \leq M$ and both work for $\mathbb{Z}_p$ and $\mathbb{Z}_{2^k}$.

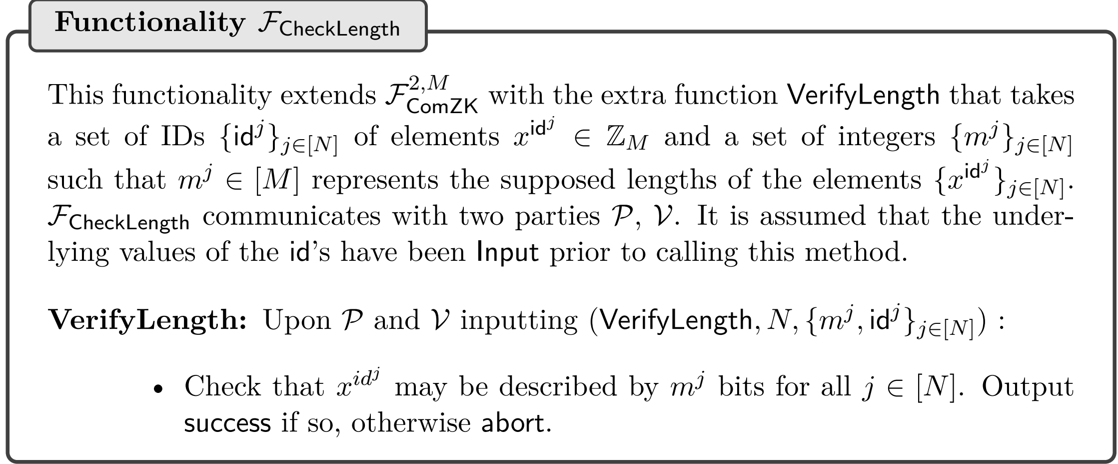
Theorem 10. *The protocol $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 3.14) UC-realizes $\mathcal{F}_{\mathsf{VerifyTrunc}}$ (Figure 3.11) in the $\mathcal{F}_{\mathsf{CheckLength}}$-hybrid model.*

Before giving 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 $[\![a']\!] = [\![a \mod 2^m]\!]$ and $[\![a_{tr}]\!]$, then when both of these are subtracted from $[\![a]\!]$, then it will be equal to 0 as required by CheckZero.

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

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

CheckZero$([\![a]\!] - (2^m \cdot [\![a_{tr}]\!] + [\![a']\!]))$: This check ensures correctness of the two values $[\![a']\!]$ and $[\![a_{tr}]\!]$. 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.



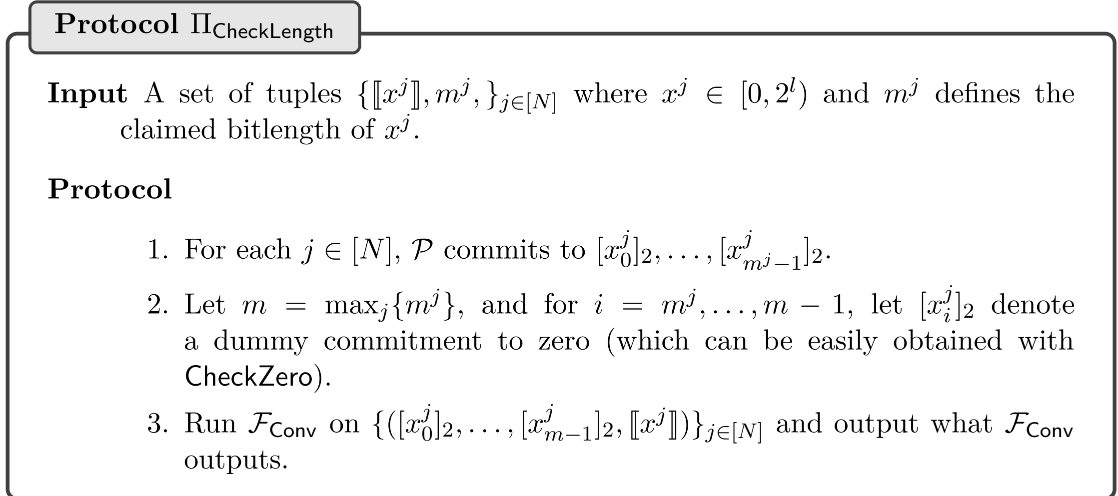
Functionality $\mathcal{F}_{\mathsf{CheckLength}}$

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

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

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

Figure 3.12: Functionality to verify length of commitments



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

Input A set of tuples $\{[\![x^j]\!], m^j,\}_{j\in[N]}$ 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 $[x^j_0]_2, \ldots, [x^j_{m^j-1}]_2$.
2. Let $m = \max_j\{m^j\}$, and for $i = m^j, \ldots, m-1$, let $[x^j_i]_2$ denote a dummy commitment to zero (which can be easily obtained with CheckZero).
3. Run $\mathcal{F}_{\mathsf{Conv}}$ on $\{([x^j_0]_2, \ldots, [x^j_{m-1}]_2, [\![x^j]\!])\}_{j\in[N]}$ and output what $\mathcal{F}_{\mathsf{Conv}}$ outputs.

Figure 3.13: Protocol $\Pi_{\mathsf{CheckLength}}$ that verifies that committed elements are bounded.

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

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

Protocol

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

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

Figure 3.14: Protocol to verify the truncation of an element from $\mathbb{Z}_M$

We now proceed with the proof.

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

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

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

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 $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}_{\mathsf{VerifyTrunc}}$. It

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

1. $\mathcal{S}$ forwards the call $(\mathsf{VerifyTrunc}, N, \{m^j, [\![a^j]\!], [\![a^j_{tr}]\!]\}_{j\in[N]})$. If $\mathcal{F}_{\mathsf{VerifyTrunc}}$ returns $\perp$, output $\perp$ to $\mathcal{V}^*$ and abort.

2. For each $j \in [N]$ $\mathcal{S}$ commits to a random value $[\![a']\!]$ using Input of $\mathcal{F}_{\mathsf{CheckLength}}$. We assume that simulated commitments to $a^j, a^{',j}$ already exist in $\mathcal{F}_{\mathsf{CheckLength}}$.

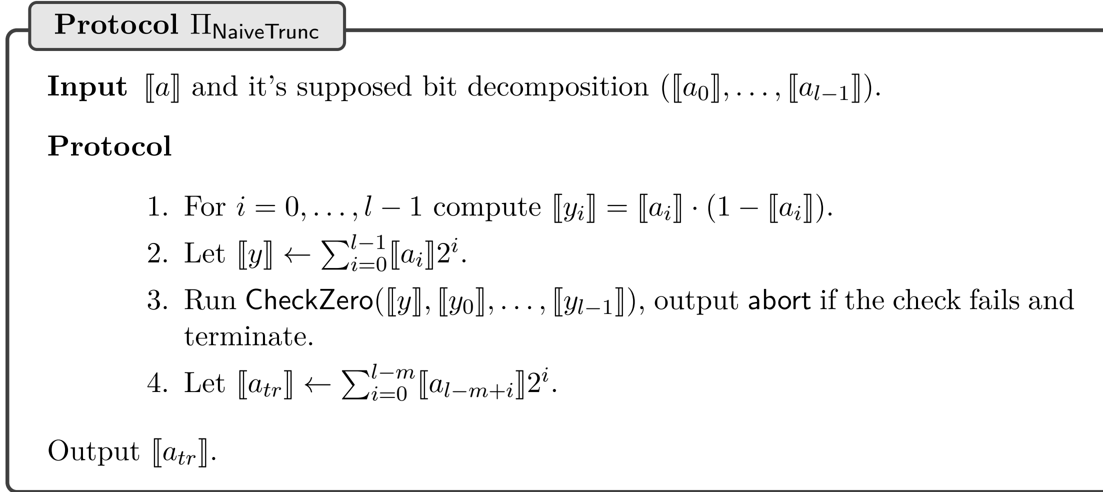
3. For each iteration $j \in [N]$, let l be the size of a^j and m be the size of $a^{',j}$. $\mathcal{S}$ runs $(\mathsf{CheckLength}, \mathsf{id}^{a^{',j}}, m)$ and $(\mathsf{CheckLength}, \mathsf{id}^{a^j_{tr}}, l-m)$ in $\mathcal{F}_{\mathsf{CheckLength}}$ towards the verifier.

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

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

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 $[\![a_i]\!]$ 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 [WYKW21b] to check the multiplications (alternative protocols such as [BMRS21b] 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.



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

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

Protocol

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

Output $[\![a_{tr}]\!]$.

Figure 3.15: Protocol that naïvely truncates a by m bits

Table 3.2: Comparison of the costs of $\Pi_{\mathsf{NaiveTrunc}}$ (Figure 3.15) and $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 3.14).

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

A breakdown of the costs of $\Pi_{\mathsf{NaiveTrunc}}$ compared to those of our optimized protocol $\Pi_{\mathsf{VerifyTrunc}}$ (Figure 3.14) is given in Table 3.2, 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.

3.4.2 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 $[\![\alpha]\!] - [\![\beta]\!]$ and have the prover compute the truncation of this which is only the most significant bit. Now we may run

$\Pi_{\mathsf{VerifyTrunc}}$ on the truncation and use the truncation as the output of the comparison. We remark that, similarly to previous works in the MPC setting [Cd10, EGK+20], this gives the correct result as long as $\alpha, \beta \in [-2^{l-2}, 2^{l-2})$, so that $\alpha - \beta \in [-2^{l-1}, 2^{l-1})$, so this introduces a mild restriction on the range of values that can be supported.

3.5 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 3.5.1. Then, in Section 3.5.2, we present two protocol variants which instantiate $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$, and prove their security in Section 3.5.3.

3.5.1 Linearly Homomorphic Commitments from Vector-OLE

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

$$\mathsf{M}[x] := \Delta \cdot \tilde{x} + \mathsf{K}[x] \in \mathbb{Z}_{2^{k+s}} \tag{3.2}$$

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 $\mathsf{M}[x]$, whereas $\mathcal{V}$ receives Δ and $\mathsf{K}[x]$.

Initially $\tilde{x}$ may be chosen as $\tilde{x} = x \in \{0, \dots, 2^k - 1\}$. Applying the arithmetic operations described below can result in larger values though, which do not get reduced modulo 2^k because all computation happens modulo 2^{k+s}. For a commitment $[x]$ we always use $\tilde{x}$ to denote the representative 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 $\mathsf{M}[z] \leftarrow a \cdot \mathsf{M}[x] + \mathsf{M}[y]$, as well as $\mathsf{K}[z] \leftarrow a \cdot \mathsf{K}[x] + \mathsf{K}[y] - \Delta \cdot b$. Then we have

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

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

As in SPD$\mathbb{Z}_{2^k}$, the MACs are obtained using vector OLE over rings. We describe the protocols in the $\mathcal{F}^{s,r}_{\mathsf{vole2k}}$-hybrid model (cf. Figure 3.16); in Section 3.5.4, 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 $\tilde{r} \in_R \mathbb{Z}_{2^{k+s}}$.

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

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

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

All further Input queries are ignored.

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

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

Figure 3.16: Functionality for vOLE with key size s and message size r. Based on $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ from [WYKW21b, Fig. 2].

Then, $\tilde{z}$ and $\mathsf{M}[z]$ are published and the MAC equation (Equation (3.2)) is verified. Following [WYKW21b, DNNR17], we implement more efficient batched checks based on a random oracle in protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 3.17 & 3.18) and protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ (Figure 3.19).

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

In this section, we present two protocols $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ and $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ which instantiate the $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ functionality (Figure 3.1). These are adaptions of the Wolverine [WYKW21b] and Mac'n'Cheese [BMRS21b] 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 [WYKW21b, DNNR17] based on a random oracle $H\colon \{0,1\}^* \to \{0,1\}^s$: First, the upper s bits (resp. the upper $2s$ bits in $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$) 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 $h := H(\mathsf{M}[y_1], \ldots, \mathsf{M}[y_n])$ to $\mathcal{V}$. Finally, $\mathcal{V}$ uses the p_i to recompute the MAC tags $\mathsf{M}[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + \mathsf{K}[y_i]$ and verifies that $h \stackrel{?}{=} H(\mathsf{M}[y_1]', \ldots, \mathsf{M}[y_n]')$ holds.

A previous version of this paper used a version of the batched check described in SPD$\mathbb{Z}_{2^k}$ [CDE$^+$18] 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.

$\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 3.17 & 3.18) adapts the bucketing approach from Wolverine [WYKW21b]: Let $C, B \in \mathbb{N}$ be the parameters of the bucketing scheme. To check that a collection of triples $([a_i], [b_i], [c_i])_{i=1}^n$ satisfy a multiplicative relation, i.e. $a_i \cdot b_i = c_i$ for $i = 1, \ldots, n$, the prover creates a set of $\ell := n \cdot B + C$ unchecked multiplication triples of commitments. After randomly permuting the ℓ triples according to the choice of the verifier, C triples are opened and checked by the verifier. The remaining nB triples are evenly distributed into n buckets. Then, each multiplication $(a_i \cdot b_i \stackrel{?}{=} c_i)$ is verified with the B triples of the corresponding bucket with a variant of Beaver's multiplication trick [Bea92]. For the check to pass despite an invalid multiplication $a_i \cdot b_i \neq c_i$, the adversary needs to corrupt exactly those triples that end up in the corresponding bucket for that multiplication.

For $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ (Figure 3.19), we have adapted the multiplication check of Mac'n'Cheese [BMRS21b], which is similar to the Wolverine [WYKW21b] optimization for large fields and SPDZ-style [DPSZ12] sacrificing of multiplication triples. The soundness of this type of check is based on the difficulty of finding a solution to a randomized equation. If a multiplicative relation does not hold, the adversary needs to guess a random field element in order to pass. Thus the original scheme needs a large field to be sound. In the $\mathbb{Z}_{2^k}$-setting, there are multiple obstacles that we have to overcome. First, we would like to also support small values of k (e.g. $k = 8$ or 16). Simultaneously, we also have to deal with zero divisors (which complicate the check) which were no issue in the field setting. Moreover, even though the commitment scheme (see Section 3.5.1) 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 $\tilde{x}, \mathsf{M}[x], \mathsf{K}[x]$ now require $k + 2s$ bits. We discuss the communication complexity of both variants in Section 3.6.1.

3.5.3 Proofs of Security

In this section we formally state the security guarantees of our protocols. and give an overview of the corresponding proofs.

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

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

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

Random For (Random), the parties send (Extend) to $\mathcal{F}^{s,k+s}_{\mathsf{vole2k}}$ so that $\mathcal{P}$ receives $\mathsf{M}[r], r \in \mathbb{Z}_{2^{k+s}}$ and $\mathcal{V}$ receives $\mathsf{K}[r] \in \mathbb{Z}_{2^{k+s}}$ so that $\mathsf{M}[r] = \Delta \cdot r + \mathsf{K}[r]$ 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 $\mathcal{P}$),
- $\mathsf{M}[z] \leftarrow \sum_{i=1}^n \alpha_i \cdot \mathsf{M}[x_i]$ (by $\mathcal{P}$),
- $\mathsf{K}[z] \leftarrow -\Delta \cdot \alpha_0 + \sum_{i=1}^n \alpha_i \cdot \mathsf{K}[x_i]$ (by $\mathcal{V}$).

CheckZero Let $H\colon \{0,1\}^* \rightarrow \{0,1\}^s$ 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 $[r_1], \ldots, [r_n] \leftarrow \mathsf{Random}()$ 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 $h \leftarrow H(\mathsf{M}[y_1], \ldots, \mathsf{M}[y_n])$ and sends $h \in \{0,1\}^{2\lambda}$ to the verifier.
5. Finally, $\mathcal{V}$ computes $\mathsf{M}[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + \mathsf{K}[y_i] \in \mathbb{Z}_{2^{k+s}}$ for $i = 1, \ldots, n$, checks $h \stackrel{?}{=} H(\mathsf{M}[y_1]', \ldots, \mathsf{M}[y_n]')$ 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 $[r] \leftarrow \mathsf{Random}()$. 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 3.17: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ using a Wolverine-like [WYKW21b] multiplication check.

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

MultiplicationCheck Let $B, C \in \mathbb{N}$ be parameters of the protocol. On input $(\mathsf{CheckMult}, ([a_i], [b_i], [c_i])_{i=1}^{n})$ 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 $\mathsf{lst} \leftarrow \emptyset$
3. They compute $([x_i], [y_i])_{i=1}^{\ell} \leftarrow \mathsf{Random}()$ 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 $([z_i])_{i=1}^{\ell} \leftarrow \mathsf{Input}((z_i)_{i=1}^{\ell})$.
5. $\mathcal{V}$ samples a permutation $\pi \in_R S_\ell$ and sends it to $\mathcal{P}$.
6. They run $(x_{\pi(i)}, y_{\pi(i)}, z_{\pi(i)})_{i=1}^{C} \leftarrow \mathsf{Open}(([x_{\pi(i)}], [y_{\pi(i)}], [z_{\pi(i)}])_{i=1}^{C}, \mathsf{lst})$.
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) $d \leftarrow \mathsf{Open}([a_j] - [x_{\pi(k)}], \mathsf{lst})$ and $e \leftarrow \mathsf{Open}([b_j] - [y_{\pi(k)}], \mathsf{lst})$ 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 $(\mathsf{CheckZero}, \mathsf{lst}, ([w_k])_{k=C+1}^{\ell})$. If successful and the check in Step 7 also passed, $\mathcal{V}$ outputs $(\mathsf{success})$ and aborts otherwise.

Figure 3.18: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ using a Wolverine-like [WYKW21b] multiplication check.

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

Much of the protocol is identical to $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 3.17 and 3.18) with the exception that the MACs are now computed in the larger ring $\mathbb{Z}_{2^{k+2s}}$. **Init**, **Random**, **Affine Combination**, **Input** and **Open** work exactly as in $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$, although using $\mathcal{F}^{s,k+2s}_{\mathsf{vole2k}}$.

CheckZero Let $H\colon \{0,1\}^* \to \{0,1\}^s$ denote a random oracle. For $(\mathsf{CheckZero}, [x_1], \dots, [x_n])$, the parties proceed as follows:

1. If one of the x_i is not equal to 0, then $\mathcal{P}$ aborts.
2. They run $[r_1], \dots, [r_n] \leftarrow \mathsf{Random}()$ and compute $[y_i] \leftarrow [x_i] + 2^k \cdot [r_i]$ for $i = 1, \dots, n$.
3. $\mathcal{P}$ sends $p_1, \dots, p_n$ 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 $h \leftarrow H(\mathsf{M}[y_1], \dots, \mathsf{M}[y_n])$ and sends $h \in \{0,1\}^{2\lambda}$ to the verifier.
5. Finally, $\mathcal{V}$ computes $\mathsf{M}[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + \mathsf{K}[y_i] \in \mathbb{Z}_{2^{k+2s}}$ for $i = 1, \dots, n$, checks $h \stackrel{?}{=} H(\mathsf{M}[y_1]', \dots, \mathsf{M}[y_n]')$ and outputs $(\mathsf{success})$ if the equality holds and aborts otherwise

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

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

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

Figure 3.19: Protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ instantiating $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ using a Mac'n'Cheese-style [BMRS21b] multiplication check.

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

We state the security of our protocol as follows:

Theorem 11. *The protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ (Figures 3.17 & 3.18) securely realizes the functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$: No environment can distinguish the real execution from a simulated one except with probability $(q_{\mathsf{cz}} + q_{\mathsf{cm}}) \cdot 2^{-s+1} + q_{\mathsf{cm}} \cdot \binom{nB+C}{B}^{-1}$, 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 $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$. These are stated in the following two lemmata. The proof of Theorem 11 is given afterwards.

Lemma 10. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckZero *protocol of $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{com-a}}$ with commitments $[x_1], \dots, [x_n]$ and $x_i \not\equiv_k 0$ for some $i \in \{1, \dots, n\}$ then $\mathcal{V}$ outputs* (success) *with probability at most $\varepsilon_{\mathsf{cz}} := 2^{-s+1}$.*

Proof of Lemma 10. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the protocol on commitments $[x_1], \dots, [x_n]$, but $(x_1, \dots, x_n) \not\equiv_k (0, \dots, 0)$. Hence, there is an index $i^* \in \{1, \dots, n\}$ 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 $i = 1, \dots, n$ denote the values sent by the prover instead of $p_1, \dots, p_n$ and define $a := (\Delta \cdot 2^k \cdot p'_1 + \mathsf{K}[[]y_1], \dots, \Delta \cdot 2^k \cdot p'_n + \mathsf{K}[[]y_n])$. 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 $\mathsf{M}[[]y_{i^*}] = \Delta \cdot \tilde{y}_{i^*} + \mathsf{K}[[]y_{i^*}]$, $\mathcal{P}^*$ needs to come up with a value $\mathsf{M}[[]y_{i^*}]' = \Delta \cdot 2^k \cdot p'_{i^*} + K[y_{i^*}]$. 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}. □

Note that the CheckZero protocol is based on the batch check from [WYKW21b, DNNR17],

Lemma 11. *If $\mathcal{P}^*$ and $\mathcal{V}$ run the* CheckMult *protocol of* $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{com-a}}$ *with parameters* $B, C \in \mathbb{N}$ *such that* $C \geq B$ *and inputs* $([a_i],[b_i],[c_i])_{i=1}^n$ *and there exists an index* $1 \leq i \leq n$ *such that* $a_i \cdot b_i \not\equiv_k c_i$ *then* $\mathcal{V}$ *outputs* (success) *with probability at most* $\varepsilon_{\mathsf{cm}} + \varepsilon_{\mathsf{cz}}$ *with* $\varepsilon_{\mathsf{cm}} := \binom{nB+C}{B}^{-1}$, *and* $\varepsilon_{\mathsf{cz}}$ *the soundness error of* CheckZero *given in Lemma 10.*

Proof of Lemma 11. 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 [Bea92]. 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 10). 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. [WYKW21b] have formalized this as a “balls and bins game”. According to Lemma 2 of [WYKW21b], 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 $\varepsilon_{\mathsf{cz}} + \varepsilon_{\mathsf{cm}}$. □

Note that the CheckMult protocol is based on the corresponding check from Wolverine [WYKW21b], and the same analysis also applies to the $\mathbb{Z}_{2^k}$ case.

We now state the proof of Theorem 11.

Proof of Theorem 11. To show security in the UC-model, we construct a simulator $\mathcal{S}$ with access to the ideal functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. 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 $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. It also creates copies of the prover $\mathcal{P}^*$ and verifier $\mathcal{V}$, and runs the verifier according to the protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$, while letting the prover behave as instructed by the environment. For this, $\mathcal{S}$ simulates the functionality of $\mathcal{F}^{s,k+s}_{\mathsf{vole2k}}$ with corrupted $\mathcal{P}$. If the simulated $\mathcal{P}$ aborts the protocol, $\mathcal{S}$ sends (abort) to $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. The method calls are simulated as follows:

For Random, the parties call the Expand of $\mathcal{F}^{s,k+s}_{\mathsf{vole2k}}$ to generate a commitment $[r]$ of the form $\mathsf{M}[r] = \Delta \cdot \hat{r} + \mathsf{K}[r]$. Since, $\mathcal{P}^*$ is corrupted, it is allowed to choose its outputs

$\hat{r}, \mathsf{M}[r] \in \mathbb{Z}_{2^{k+s}}$. $\mathcal{S}$ 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 $\hat{r}, \mathsf{M}[r] \in_R \mathbb{Z}_{2^k+s}$ 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 $(\mathsf{success})$ in the real execution, then it does the same in the simulation except with probability at most $\varepsilon_{\mathsf{cz}}$ (defined in Lemma 10). 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 10, 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 11, and get that the output of $\mathcal{V}$ is the same in the simulation and in the real execution except with probability at most $\varepsilon_{\mathsf{cz}} + \varepsilon_{\mathsf{cm}}$.

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

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

Malicious Verifier The setup of the simulation in case of a corrupted verifier $\mathcal{V}^*$ is similar as before. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{V})$ to the ideal functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. It creates copies of the prover $\mathcal{P}$ and verifier $\mathcal{V}^*$. The prover is run according to the protocol, whereas the environment controls the verifier. For this, $\mathcal{S}$ simulates the functionality of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ with corrupted $\mathcal{V}$. For all methods, since $\mathcal{V}$ does not have any private inputs no

input extraction is necessary. So the simulator can just send the corresponding message on behalf of the verifier to $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$. The method calls are simulated as follows:

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

For Random, the parties call the Expand of $\mathcal{F}_{\mathsf{vole2k}}^{s,k+s}$ to generate a commitment $[r]$ of the form $\mathsf{M}[r] = \Delta \cdot \hat{r} + \mathsf{K}[r]$ where $\mathcal{V}^*$ can choose $\mathsf{K}[r]$. $\mathcal{S}$ 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 $\Delta, \mathsf{K}[x_1], \ldots, \mathsf{K}[x_n], \mathsf{K}[r_1], \ldots, \mathsf{K}[r_n]$, so we can sample $p_1, \ldots, p_n \in_R \mathbb{Z}_{2^s}$ and compute $\mathsf{M}[y_i]' \leftarrow \Delta \cdot 2^k \cdot p_i + \mathsf{K}[x_i] + 2^k \cdot \mathsf{K}[r_i]$ for $i = 1, \ldots, n$. Then, the p_i and $h := H(\mathsf{M}[y_1]', \ldots, \mathsf{M}[y_n]')$ 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. □

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

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

Theorem 12. *The protocol $\Pi_{\mathsf{ComZK\text{-}b}}^{\mathbb{Z}_{2^k}}$ (Figure 3.19) securely realizes the functionality $\mathcal{F}_{\mathsf{ComZK}}^{\mathbb{Z}_{2^k}}$: No environment can distinguish the real execution from a simulated one except*

with probability $(q_{\mathsf{cz}} + q_{\mathsf{cm}}) \cdot 2^{-s+1} + q_{\mathsf{cm}} \cdot 2^{-s}$*, where* q_{cz} *is the sum of calls to* CheckZero *and* Open*, and* q_{cm} *the number of calls to* CheckMult*.*

Proof of Theorem 12. Since most of $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ is actually identical to $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ we will refer to the Proof of Theorem 11 for these parts, and focus on the differences here.

The subroutines CheckZero and CheckZero′ are only very slightly modified from the CheckZero from $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$. The latter is exactly the same as in before, but for the larger message space $\mathbb{Z}_{2^{k+s}}$, and the former additionally hides some more bits. Hence, the same Lemma 10 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 11, i.e. $\mathcal{S}$ sends $(\mathsf{corrupted}, \mathcal{P})$ to the ideal functionality $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$ and simulates copies of prover and verifier.

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

If the simulated verifier aborts, it sends (abort) to $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. Thus, if the verifier aborts in the real execution, then it does the same in the simulation. On the other hand, if the verifier aborts in the simulation, then by Lemma 12 it also aborts in the real protocol, except with probability $\varepsilon_{\mathsf{cz}} + \varepsilon'_{\mathsf{cm}}$.

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

For CheckMult, we use the same strategy as in the Proof of Theorem 11: $\mathcal{S}$ sends the corresponding message on behalf of the corrupted verifier to $\mathcal{F}^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK}}$. If it aborts, then $\mathcal{S}$ instructs the simulated $\mathcal{P}$ to also abort by sending (abort) to the simulated $\mathcal{V}$. Otherwise $\mathcal{S}$ simulates the complete protocol using the constant value 0 for all of the prover's commitments so that the verifier's view is the same as in the real execution.

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

Except for the simulation of CheckMult, the proof of Theorem 12 is largely similar to the proof of Theorem 11. Again, we first prove a lemma about the soundness error of the CheckMult operation, that we use to show indistinguishability of our simulation. The proof is included here, since it shows why the adaption of the SPDZ-style sacrificing check from large fields to the $\mathbb{Z}_{2^k}$ setting is secure.

Lemma 12. *If* $\mathcal{P}^*$ *and* $\mathcal{V}$ *run the* CheckMult *protocol of* $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ *with inputs* $([a_i], [b_i], [c_i])_{i=1}^n$ *such that there exists an index* $1 \leq i \leq n$ *such that* $a_i \cdot b_i \not\equiv_k c_i$*, then* $\mathcal{V}$ *outputs* $(\mathsf{success})$ *with probability at most* $\varepsilon'_{\mathsf{cm}} + \varepsilon_{\mathsf{cz}}$ *with* $\varepsilon'_{\mathsf{cm}} := 2^{-s}$*, and* $\varepsilon_{\mathsf{cz}}$ *the soundness error of* CheckZero *given in Lemma 10.*

Proof of Lemma 12. Suppose $\mathcal{P}^*$ and $\mathcal{V}$ run the CheckMult protocol with inputs as described in the lemma. Since CheckZero′ is a variant of CheckZero from $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ for the larger message space $\mathbb{Z}_{2^{k+s}}$, we can apply Lemma 10 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

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

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

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

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 $\varepsilon_{\mathsf{cz}} + 2^{-s}$. □

3.5.4 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 [BCG+19a, WYKW21b], which currently only have an actively secure setup protocol over fields. We consider two possible alternatives. First, as done in [CDE+18], we can use the protocol for correlated oblivious transfer over general rings from [Sch18], 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 [SGRR19], or dual-LPN based on quasi-cyclic codes, as used over $\mathbb{Z}_2$ in [BCG+19a] (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 [BCG+19a], 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 [HSS17] combined with LPN-based OT [BCG+19a, YWL+20],

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 3.3: 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 $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$.

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

3.6 Evaluation

3.6.1 Communication Complexity

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

In the protocol $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$, $\mathcal{V}$ samples a permutation π in $\mathsf{CheckMult}$ 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 $\mathsf{CheckMult}$.

As described in Section 3.5.1 and Section 3.5.2, we need to randomize the upper s or $2s$ bits when doing a $\mathsf{CheckZero}$ or Open operation. We note that in $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ Step 8a and $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ 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 3.3.

Verifying Conversions

The amortized costs for verifying the correctness of a single conversion tuple $([x_0]_2, \dots, [x_{m-1}]_2, [\![x]\!])$ are given in Table 3.4, 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. In Section 3.6.2, we give a more detailed breakdown of these costs, including complexities of the sub-protocols $\mathsf{bitADDcarry}$ and $\mathsf{convertBit2A}$.

The "naïve" way of verifying the a conversion would be to have the prover provide both a set of bits $\chi = \{[x_0]_p, \dots, [x_{m-1}]_p\}$ 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 $\mathsf{CheckZero}([x_i]_p \cdot ([x_i]_p - 1)), [x_i]_p \in \chi$ 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 3.4. To verify

Table 3.4: 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 3.3" uses our optimizations. "QS-Circuit" and "QS-Poly" refer to variants that use [YSWW21b] 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 $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}a}}$ are given. The $\Pi^{\mathbb{Z}_{2^k}}_{\mathsf{ComZK\text{-}b}}$ variant requires Bs additional bits of communication (see Section 3.5.2).

1 Protocol	1 Comm. in bits	1 #COTs	1 #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 3.3, $\mathbb{Z}_p$	$8Bm + B$	$4Bm + B$	$2B$
Section 3.3, $\mathbb{Z}_{2^k}$	$5Bm + Bs - 3B$	$4Bm - 3B$	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

the multiplications we use the basic version of Mac'n'Cheese [BMRS21b]. The "basic" baseline comparison in Table 3.4 comes from a straightforward application of using `edaBits` for ZK, similarly to [EGK+20]. Namely, this protocol would first generate consistent `edaBits` using [EGK+20], and then verify the conversion using a single binary addition circuit (similar to the bucket-check in Figure 3.5, step 6). However, this requires doing the check with m *verified* multiplication triples (over $\mathbb{Z}_2$) and a single `daBit`, which in turn requires an additional verified multiplication (over $\mathbb{Z}_M$). To estimate these costs, we used [WYKW21b] for verifying AND gates at a cost of 7 bits per gate, and [BMRS21b] for verifying triples in a larger field.

Since COTs and VOLEs can be obtained from pseudorandom correlation generators with very little communication [YWL+20, WYKW21b], the remaining online communication dominates. Hence, our optimized protocol from Section 3.3 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 945 bits, a reduction of around 35%.

Note that, as mentioned in Section 3.1.2, the "basic" approach can be optimized by verifying multiplications with QuickSilver [YSWW21b] or the amortized version of Mac'n'Cheese [BMRS21b]. 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 $\mathsf{bitADDcarry}$ can be either rep-

resented 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 [WYX+21b]) 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).

3.6.2 Sub-protocols

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

Complexity of bitADDcarry

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

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, \dots, m-1\} \tag{3.3}$$

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

$$z_i = x_i \oplus y_i \oplus c_i, \forall i \in \{0, \dots, m-1\} \tag{3.4}$$

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

Complexity of convertBit2A

We consider the procedure convertBit2A as defined in Figure 3.6. We assume that the input (not the `daBit`) is distributed prior to running the protocol. This sub-protocol requires a single `daBit` to convert the bit authenticated in $\mathbb{F}_2$ to $\mathbb{F}_M$. Having a single `daBit` $([r]_2, [r]_M)$, we can convert a value $[x_m]_2$ by following the following protocol.

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

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

Table 3.5: 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	$\geq$1 048 576
40	4	$\geq$ 10 322
40	5	$\geq$ 1 024
80	5	$\geq$1 048 576

3.6.3 Experiments

We have implemented the conversion protocol Π_{Conv} 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 [WYKW21b]. All benchmarks were run on a MacBook Pro 2018, 2.9 GHz 6-Core Intel Core i9, 32 GB 2 400 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.

Experimental Results

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 3.5. Lastly, all of these listed benchmarks are run using Wolverine [WYKW21b] to verify multiplications as in protocol Π_{Conv} of Section 3.3.2. We also implemented the variant of our protocol using QuickSilver [YSWW21b], which reduces the runtimes by around a factor of two, thanks to the lower communication and preprocessing requirements.

We benchmarked our conversion protocol Π_{Conv} from Section 3.3, as well as a variant which uses Quicksilver [YSWW21b] to verify the multiplications in $\mathsf{bitADDcarry}$ (instead of faulty multiplication triples). We run Π_{Conv} to verify $N = 1\,024,\ 10\,322,\ 1\,048\,576$ 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 3.6 and 3.7 show the measured communication and run-times 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 3.8 and 3.9 show the measured communication and run-times for the variant with QuickSilver [YSWW21b].

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

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

Table 3.6: 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 3.3.

	2 $m = 8$	2 $m = 16$	2 $m = 32$	2 $m = 60$				
	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$
$N = 1\,024$, $B = C = 5$								
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	1 488.70	57.77	2 522.08	111.09	4 591.06	222.17		

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 3.10 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 3.11 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.

Table 3.7: Run-time in s when verifying N conversion tuples of bit size m with bucket size $B = C$ using our protocol from Section 3.3.

m		20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s
$N = 1\,024,\ B = C = 5$						
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 3.8: 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 [YSWW21b] to verify multiplications.

	2 $m = 8$	2 $m = 16$	2 $m = 32$	2 $m = 60$				
	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$	$\mathcal{P}$	$\mathcal{V}$
$N = 1\,024$, $B = C = 5$								
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	1 282.92	48.88	2 108.28	88.86		

Table 3.9: Run-time in s when verifying N conversion tuples of bit size m with bucket size $B = C$ using QuickSilver [YSWW21b] to verify multiplications.

m		20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s
$N = 1\,024,\ B = C = 5$						
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

Table 3.10: Communication between the prover (P) and verifier (V) required to verify 2^{20} conversion tuples where $C = B = 3$ with Wolverine [WYKW21b] multiplication check.

bit size (m)	2 8		2 16		2 32	
	P	V	P	V	P	V
init (in Mb)	100.1	10.8	100.1	10.8	100.1	10.8
input (in Mb)	71.7	8.9	73.9	13.4	78.3	22.3
conv (in Mb)	1488.7	57.8	2522.1	111.1	4591.1	222.2

Table 3.11: Run-time in s for verifying 2^{20} conversion tuples with $m = 32$ and $B = C = 3$ with multiplication check of [WYKW21b].

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

Chapter 4

Cheddar – Oh, Range Proofs for VOLE-based Zero-Knowledge!

4.1 Introduction

Zero-Knowledge (ZK) proofs allow a prover to prove to a verifier that a certain statement is true, while revealing no information beyond that the statement is true. The verifier should only be convinced of statements that are true, regardless of whether or not the prover is behaving honestly. This idea works for arbitrary functions where the computation typically is modeled as circuits of operations and the efficiency of a proof depends on the number of gates that the circuit has. Rather than use general circuits for everything, oftentimes protocols built for specific purposes tend to be more efficient in terms of communication and computation. One of these specific types of zero-knowledge proofs are those that allows a prover to prove that a secret value x (committed or encrypted) lies in some public range $[a, b]$, meaning that $a \leq x \leq b$. This is what is known as a zero-knowledge *range proof*.

Range proofs are a core component in many applications including anonymous credentials [Cha90], e-voting [Gro05, ABG+21], e-cash [CHL05], or lattice-based cryptography in general where security and correctness is based on the smallness of certain integers and vectors [CMM19, LLM+16, GHL22]. Also of interest in recent years is verifiability of large-scale neural network inference [LXZ21]. This requires a large number of both linear operations as well as non-linear operations such as ReLU, Sigmoid, Max Pooling, SoftMax and the likes. The main difficulty here lies in the fact that these non-linear operations cause a loss in accuracy when the values are represented as fixed-point numbers, something that the linear operations require. To this end, range proofs can also be used to convert between fixed-point numbers and floating-point numbers, which can then be utilized in the protocols to obtain maximum accuracy.

As seen above, range proofs have a wide variety of applications and due to this, many constructions have been proposed. What all of these have in common, is that they fall into two main paradigms: (1): range proofs based on n-ary decomposition [Gro11, CCs08, BBB+18], or (2): range proofs based on square decomposition [Bou00, Lip03, Gro05, CPP17, GHL22].

(1) When considering proves based on n-ary decomposition, the prover will prove statements of the form $x \in [0, n^k)$ by computing and committing to an n-ary decomposition $(x_0, \dots, x_{k-1})$ of x. The prover then has to show that $x = \sum_{i=0}^{k-1} x_i \cdot n^i$ as well as $x_i \in [0, n)$. If the parties are using a commitment scheme satisfying some homomorphic properties, the above approach can usually be made into a generic proof of $x \in [a, b]$ instead. In this paradigm, the leading method is Bul-

letproofs [BBB+18] whose approach uses generalized Pedersen commitments to commit to the bit-decomposition of x as well as an efficient way of proving that all the committed values are bits. A usual advantage of proofs in this paradigm (one also Bulletproofs satisfy) is that they won't need to rely on trusted setups, which comes in handy when considering real-world applications such as cryptocurrencies, where it is critical that there exists no central authority. Constructions based on this paradigm however, require more commitments based on the size of the ranges as well as n, which can be quite costly in communication.

Recently, a third alternative [BBMH+21b] was proposed, which allows the prover to commit to the decomposition in a binary field rather than the large field which x exists in. This saves communication, as the keys and elements are much smaller and it doesn't require the prover to prove that the committed-to elements are bits. It puts no restriction on the types of field.

(2) When considered proves based on square decomposition, the prover initially have some x where the prover wants to prove that $x \in [a, b]$. This reduces to proving that $x-a$ and $b-x$ are positive, which in turns means that $y = (x-a)(b-x)$ is positive (unless both are negative of course). Proving that y is positive, can be done by having the prover compute four integers $(\alpha, \beta, \gamma, \delta)$ so that $y = \alpha^2 + \beta^2 + \gamma^2 + \delta^2$. The idea of showing that an integer is positive by decomposing it into four squares was initially proposed by Lipmaa [Lip03]. This can always be done (assuming y is positive), due to Lagrange's four square theorem. The prover then commits to these four squares and shows that $(\alpha, \beta, \gamma, \delta)$ is a correct decomposition (i.e. $y = \alpha^2 + \beta^2 + \gamma^2 + \delta^2$) over the integers. It is crucial that this is over the integers, as any overflow in this computation can cause an incorrect result. Thus, protocols based on this paradigm require only a fixed number of commitments. Adding further to this particular strength, [Gro05] noticed that one can decompose $4y + 1$ as a sum of three squares rather than 4 (positive integers congruent to 1 mod 4 can always be decomposed like this). This comes at the cost of leaking 1 bit for each element.

All (until recently) known integer commitment schemes require the use of RSA groups or large class groups, which both require trusted setups. In a recent work [CKLR21] however, this was done away with by constructing a bounded integer commitment scheme, which is based on groups in which the discrete log (DLOG) problem is hard. The main drawback though of this construction, is that it requires quite large groups which are not used in standard applications.

4.2 Square Decomposition

In this section we will first describe how to compute both a four-square decomposition as well as a three-square decomposition. We then discuss a potential optimization to the way of computing the three-square decomposition; as part of the protocol the prover has to factor a random integer into its prime factors. If the random integer is a semiprime,

this will succeed. We instead consider a different strategy in which we sample the two prime numbers first.

4.2.1 Four-Square Decomposition

To write x as four-squares, one can use the following algorithm due to Lipmma [Lip03].

1. Write x in the form $x = 2^t(2k+1)$ where $t, k \geq 0$.
2. If $t = 1$ then:
   (a) Choose random $a \leq \sqrt{x}$, $b \leq \sqrt{x - a^2}$, such that exactly one of a, b is even. Let $p \leftarrow x - a^2 - b^2$. Now $p \equiv 1 \bmod 4$.
   (b) If p is not prime, then go back to step 2a. Otherwise use Cornacchias algorithm (Section 4.2.3) to compute $p = c^2 + d^2$.
   (c) Output a, b, c, d
3. If t is odd (but not 1), find a representation (a, b, c, d). Let $s \leftarrow 2^{(t-1)/2}$ and output (sa, sb, sc, sd).
4. If t is even, find a representation $a^2 + b^2 + c^2 + d^2$ for $x = 2(2k + 1)$ instead, using Step 2. Then convert this representation to a representation for $x = (2k+1)$ instead as follows: Group a, b, c, d so that $a \equiv b \bmod 2$ and $c \equiv d \bmod 2$. Let $s \leftarrow 2^{(t-1)/2}$ and output $(s(a + b), s(a - b), s(c + d), s(c - d))$.

Which leads to the following theorem

Theorem 13 (four-square decomposition)**.** *An integer x can be represented as $x = a^2 + b^2 + c^2 + d^2$ with integers a, b, c, d if and only if $x \geq 0$. Moreover, if $x \geq 0$, then the corresponding representation can be computed efficiently by using the above algorithm.*

A proof of the above theorem is due to Lipmaa [Lip03].

4.2.2 Three-Square Decomposition

In this section we will first describe the original algorithm for computing a three-square decomposition by Rabin and Shallit[RS86], before describing an optimized variation due to Groth [Gro05] which revolves around factoring. Finally we will discuss probabilities of whether or not random integers are perfect squares or semiprimes, as part of a potential optimization to the algorithm due to Groth.

Original Three-Square Decomposition

This algorithm is due to Rabin and Shallit[RS86].

1. If $n \equiv 0 \bmod 4$: Compute $(x, y, z) \leftarrow three - squares(\frac{n}{4})$ and output $2x, 2y, 2z)$

2. If $n \equiv 7 \bmod 8$: Output “No representation” and abort

3. If $n \equiv 3 \bmod 8$:

   (a) Choose random $x \leq \lfloor \sqrt{n} \rfloor$
   (b) Compute $p \leftarrow \frac{1}{2}(n - x^2)$
   (c) If p is a prime, let $(y, z) \leftarrow \mathit{cornacchia}(p)$ and output $(x, y + z, y - z)$. Otherwise go to Step 3a

4. If n is a perfect square: then output $(\sqrt{n}, 0, 0)$

5. If $n \equiv 1 \bmod 4$ or $n \equiv 2 \bmod 4$:

   (a) Choose random $x \leq \lfloor \sqrt{n} \rfloor$
   (b) Compute $p \leftarrow (n - x^2)$
   (c) If p is a prime, let $(y, z) \leftarrow \mathit{cornacchia}(p)$ and output (x, y, z). Otherwise go to Step 5a

Theorem 14. *The preceding algorithm is correct and with high probability returns an expression of n as the sum of three squares, for all sufficiently large n.*

The proof of Theorem 14 can be found in [RS86]. The above algorithm has been altered slightly from the original. Specifically, we use cornacchia's algorithm to compute the two-square decomposition of primes. This algorithm is faster than what Rabin and Shallit originally proposed and both algorithms rely on Theorem 15. Note that the above algorithm could potentially return "*No representation*" on some integers, which could leak information on the provers input, in case this is used during a range proof. A fix for this would be to compute a new input $n \leftarrow 4n + 1$ based on the original input n. Now, if this new n is positive, then so is the original n. This would *force* a nice representation, such that it can always be decomposed, at the cost of working over a slightly larger input.

Three-Square Decomposition via Factoring

One can also compute a three-square decomposition by factoring an integer into two prime factors. Groth [Gro05] proposes a way of decompositing $4x + 1$ into three squares to show that $4x+1$ is positive (thus also showing that x is), such that $4x+1 = a^2+b^2+c^2$. The strategy due to Groth is

1. Guess an even a_i at random such that $4x + 1 - a^2$ is a product of primes on the form $X \cdot Y \equiv 1 \bmod 4$ such that $X \equiv 1 \bmod 4$ and $Y \equiv 1 \bmod 4$

2. Write each such prime as the sum of two squares: $(e, r) \leftarrow \mathit{cornacchia}(X)$ and $(k, l) \leftarrow \mathit{cornnachia}(Y)$ using Section 4.2.3, i.e. $X = (e^2 + r^2)$ and $Y = k^2 + l^2$.

3. Let $XY = (ek + rl)^2 + (el - rk)^2$ and $b = (ek + rl)$ and $c = (el - rk)$ so that $4x + 1 - a^2 = b^2 + c^2$. Output (a, b, c)

Note that while this idea saves communicating 1 field element, it requires prime factorization to be possible in an efficient manner.

Picking the two primes before the square

As part of the algorithm in Section 4.2.2, one first picks a random a and subtract it's square from the input $p \leftarrow 4x + 1 - a^2$. One then factor p into its prime factors and hope that the integer p is a semiprime. If this is not the case, one picks a new a and try again. The factoring step dominates the cost whenever the integers become large. One might then consider sampling two random primes, compute their product and check if this is a perfect square instead:

1. Let $p = 4x + 1$
2. Guess two random primes X, Y such that $XY \leq p$. Compute $p' = p - XY$.
3. If p' is a perfect square (i.e. is $\sqrt{p'}$ an integer), then let $a \leftarrow \sqrt{p'}$. Otherwise go to step 2.
4. Write X and Y as the sum of two squares: $(e, r) \leftarrow \mathit{cornacchia}(X)$ and $(k, l) \leftarrow \mathit{cornnachia}(Y)$ using Section 4.2.3, i.e. $X = (e^2 + r^2)$ and $Y = k^2 + l^2$.
5. Let $XY = (ek + rl)^2 + (el - rk)^2$ and $b = (ek + rl)$ and $c = (el - rk)$ so that $4x + 1 - a^2 = b^2 + c^2$. Output (a, b, c)

It is possible to reasonable efficiently sample random primes, thus the question of whether or not this algorithm would outperform the one from Section 4.2.2, comes down to whether or not there is a higher probability of hitting a perfect square rather than a semiprime.

Informally, the number of perfect squares in the range $[0, n]$ is $\sqrt{n}$ and the number of semiprimes $\pi_2(n)$ in the same range is approximately

$$\pi_2(n) \approx \sum_{p \leq \sqrt{n}} \pi(\lfloor \frac{n}{p} \rfloor)$$

. In the latter, the summand is approximately $\frac{n}{p \log n}$, so overall this sum is approximately

$$\pi_2(n) \approx \frac{n}{\log n} \sum_{p \leq \sqrt{n}} \frac{1}{p}$$

. The sum $\sum_{p \leq \sqrt{n}} \frac{1}{p}$ is asymptotically $\log \log n$, so overall

$$\pi_2(n) \approx \frac{n \log \log n}{\log n}$$

. Now, this argument is as simple as the fact that $\sqrt{n}$ scales much slower than $\pi_2(n)$, since n is part of the product of the numerator. We can therefore conclude that there are significantly more semiprimes than perfect squares.

To conclude, despite factoring being expensive for large integers, the probability of hitting a perfect square when considering potential very large integers is much less than the probability of hitting a semiprime within a few factorizations.

4.2.3 The Algorithm of Cornacchia

Theorem 15 (Fermat-Lagrange Theorem)**.** *Every odd prime p is a sum of two squares if and only if $p \equiv 1 \bmod 4$.*

In other terms, the theorem state that any odd prime p is a sum of two squares, if and only if -1 is a quadratic residue $bmodp$.

Cornacchia's algorithm finds a solution to the more general problem

$$x^2 + dy^2 = p$$

provided that p is of the form of Theorem 15. Note that a condition for such a solution to exist, is that $-d$ must be a quadratic residue modulo p, since we must have $y \not\equiv 0 \bmod p$.

We now describe the algorithm due to Cornacchia. Let p be a prime number and d be an integer such that $0 < d < p$. This algorithm then either outputs an integer solution (x, y) to the Diophantine equation $x^2 + dy^2 = p$ or says that such a solution does not exist.

1. Compute $k \leftarrow \left(\frac{-d}{p}\right)$. If $k = -1$, then the equation has no solution.
2. Compute an integer x_0 such that $x_0^2 \equiv -d \bmod p$[1]. Let $x_0 \leftarrow x_0 + kp$ so that $p/2 < x_0 < p$. Finally, set $a \leftarrow p$, $b \leftarrow x_0$ and $l \leftarrow \lfloor \sqrt{p} \rfloor$.
3. If $b > l$, set $r \leftarrow a \bmod b$, $a \leftarrow b$, $b \leftarrow r$. Repeat until $b \leq l$.
4. If d does not divide $p - b^2$ or if $c = (p - b^2)/d$ is not the square of an integer, then the equation does not have a solution. Otherwise, output $(x, y) = (b, \sqrt{c})$.

For a proof of correctness, we refer to [MN90].

4.2.4 Range proof

A naïve range-proof using a three-square (or four-square) decomposition:

- $\mathcal{P}$ wishes to show that $x \in [a, b]$ for some commited-to value x.
    1. Let $y = (x - a)(b - x)$ and commit to y.
    2. Compute three-square decomposition $(a, b, c) \leftarrow \texttt{three} - \texttt{square}(\texttt{y})$ such that $y = a^2 + b^2 + c^2$.
    3. $\mathcal{P}$ commits to a, b, c, yielding $[a], [b], [c]$.

[1] For instance using Shank's Algorithm

4. $\mathcal{P}$ and $\mathcal{V}$ compute: $[a^2], [b^2], [c^2]$
5. $\mathcal{P}$ and $\mathcal{V}$ run $\mathsf{CheckZero}([y] - [a^2] - [b^2] - [c^2])$. Now, if CheckZero return success, then output success, otherwise abort.

Clearly, if an integer commitment scheme is used, then the above protocol is correct. It is also exceptionally cheap in communication and fairly cheap in computation, provided that the integers used are not too big for the factoring to be computabe. If one instead use the four-square decomposition, using larger integers only cost more in terms of local computations.

4.3 Approximate Range Proofs

When square decomposition is used to prove that an integer is positive, it theoretically leads to very little communication, as one can get by with as little as three additional commitments. Practically however, this technique has, up until recently, required the usage of expensive integer commitment schemes as an overflow in the squaring or addition of the squares would cause an incorrect result.

In this section we describe an alternative to using integer commitments when creating a four-square (or three-square) decompositional proof. Specifically, if the prover can just show that the individual elements of the decomposition $(\alpha, \beta, \gamma, \delta)$ won't overflow when squared or overflow when added together, then an integer commitment scheme won't be required. At this point, one might envision ouroboros, the snake eating its own tale, as what we describe above requires another range proof. Thus, building a range proof using square decomposition, requires a range proof. Luckily, it turns out that an *approximate* range proof is enough where there is both a probability of failure but also a probability of the value actually falling outside the initial bound by a pre-determined amount. On a more positive note though, these are much more efficient in terms of communication and computation.

The main idea is to define a specific *slack* λ, which we can prove something is off by at most. This is done by defining the bound β we *actually* need and then have the prover show that the secret input vector $\mathbf{s}$ is less than $\frac{\beta}{\lambda}$ rather than just β. This idea is not novel [Lyu08], but we briefly describe the general idea. The prover knows a secret vector $\mathbf{s}$ which the prover wants to show is below some bound β. As we want to use this idea to prove that the square decomposition works over the integers, we must ensure that no overflow takes place. Thus, $\beta = \sqrt{\frac{q-1}{2 \cdot 4}}$ where we work over the field $\mathbb{F}_q$. This follows from needing *signed* integers (thus $\frac{q-1}{2}$) as well as needing to square and add the 4 field elements. We then follow the following template:

1. $\mathcal{P}$ has a secret vector $\mathbf{s} \in \mathbb{F}_q^d$ that $\mathcal{P}$ wants to show is less than β. As the approximate proof has a slack of λ and can be off by at most λ, an honest prover must in fact have the values $s_i \in \mathbf{s}$ be less than $\frac{\beta}{\lambda}$.
2. $\mathcal{V}$ samples $\mathbf{C} \in \mathbb{F}_q^{\kappa \times d}$ and sends this to $\mathcal{P}$.
3. $\mathcal{P}$ and $\mathcal{V}$ compute $\mathbf{y} = \mathbf{C}\mathbf{s}$.
4. $\mathcal{P}$ computes an exact range proof on each $y_i \in \mathbf{y}$ showing the values are less than β

Now, when we compute $\mathbf{C}\mathbf{s}$ we actually project the high-dimension vector $\mathbf{s}$ down to a lower dimension. We can therefore use the Johnson-Lindenstrauss Lemma to argue that if we prove the infinity bound of the resulting vector $\mathbf{y} = \mathbf{C}\mathbf{s}$, then this is a tight approximation for the infinity norm of the original $\mathbf{s}$. Specifically, let $D_{-1,0,1}$ (or D for short) be defined as a distribution such that $\Pr_{c \leftarrow D}[c = 0] = 1/2$ and $\Pr_{c \leftarrow D}[c = 1] = \Pr_{c \leftarrow D}[c = 1] = 1/4$ from which we sample $\mathbf{C}$.

We present the Johnson-Lindenstrauss Lemma in Fact 1 and then use this in Corollary 1, both of which are originally stated in [GHL21]. The Fact 1 informally states that when computing the inner product of a vector $\mathbf{s}$ and a vector sampled $\mathbf{r}$ so that $\mathbf{r} \in \mathcal{N}^d$ where $\mathcal{N}^d$ is the normal distribution centered at 0 with a variance of 1, then with very low probability will the outcome be higher than 9.75 times the ℓ_2 norm of $\mathbf{s}$.

Fact 1. *Let $\mathcal{N}$ be the continuous normal distribution centered at 0 with variance 1. Then for every vector $\mathbf{s} \in \mathbb{Z}^d$ it holds that:*

$$\mathsf{Pr}_{\mathbf{r}\leftarrow\mathcal{N}^d}\left[\left|\left\langle \mathbf{s}, \frac{1}{\sqrt{2}}\mathbf{r}\right\rangle\right| > 9.75 \cdot \|\mathbf{s}\|_2\right] = \mathsf{Pr}_{y\leftarrow\mathcal{N}}\left[|y| > 9.75 \cdot \sqrt{2}\right] < 2^{-141}$$

We then use Fact 1 in the following Corollary 1 which states that the tail of the distribution on $\|C\mathbf{s}_2$ can be bounded as if the entries of $\mathbf{C}$ were chosen from the zero-mean continuous Normal distribution of the same variance. This will allow us to use Fact 1 while choosing $\mathbf{C}$ from $D^{\kappa\times d}_{-1,0,1}$.

Corollary 1 (heuristic)**.** *Let D be the distribution on $\{0, \pm 1\}$ as described above. Under the heuristic, substitution of D with $\frac{1}{\sqrt{2}}\mathcal{N}$, for every vector $\mathbf{s} \in \mathbb{Z}^d$:*

$$\mathsf{Pr}_{\mathbf{r}\leftarrow\mathcal{D}^d}\left[|\langle \mathbf{s}, \mathbf{r}\rangle| > 9.75\|\mathbf{s}\|_2\right] \lessapprox 2^{-141}$$

Which informally converts the probability of Fact 1 into one which instead uses a discrete distribution instead.

If we couple Corollary 1 with the following observation:

$$\|s\|_\infty \leq \|s\|_2 \leq \sqrt{d}\|s\|_\infty$$

we arrive at a bound of the infinity norm using Fact 1 and Corollary 1. Now, using the above bound on the infinity norm, we arrive at an overall bound of

$$9.75 \cdot \sqrt{d} \cdot \|s\|_\infty$$

when computing the inner products.

4.3.1 Proof of Smallness

We are now ready to define the proof of smallness that we will use to show that the elements of $\mathbf{s}$ are small. This is what will be used as a subroutine during our range proof, to prove smallness of the four squares computed by the prover. This allows us to not do the computation over the integers, as the four squares are proven small enough for there to occur no wraparound modulo q during the computation.

To begin with, we will describe a lemma that we will need later on, which is a slight simplification of Lemma 2.5 of [LNS21]. Lemma 13 specifically states the chance of detecting a vector of high norm by seeing a *single* large element in the resulting vector, i.e. if $\mathbf{s}$ has a large coefficient, so does $\langle \mathbf{c}, \mathbf{s}\rangle$ with probability at least 1/2. This tells us that if $\mathbf{Cs}$ has small coefficients for a vector $\mathbf{s} \in \mathbb{F}_q^d$ and a matrix $\mathbf{C} \in D^{\kappa\times d}_{-1,0,1}$, then with high probability so does $\mathbf{s}$.

Lemma 13. *Let the distribution $D_{-1,0,1}$ be defined as above, then for all $\mathbf{s} \in \mathbb{Z}_q^d$ and $\mathbf{C}$ sampled uniformly at random from $D_{-1,0,1}^{\kappa \times d}$ it holds that*

$$\mathsf{Pr}_{\mathbf{c} \leftarrow D^d}\left[\|\langle \mathbf{c}, \mathbf{s}\rangle\|_\infty < \frac{1}{2} \cdot \|\mathbf{s}\|_\infty\right] \leq \frac{1}{2} \quad \text{and} \quad \Pr_{\mathbf{C} \leftarrow D^{\kappa \times d}}\left[\|\langle \mathbf{C}, \mathbf{s}\rangle\|_\infty < \frac{1}{2} \cdot \|\mathbf{s}\|_\infty\right] \leq 2^{-\kappa}$$

Proof. Let s_i be the coefficient of $\mathbf{s}$ such that $\|s_i\|_\infty = \|\mathbf{s}\|_\infty$. We then write

$$\langle \mathbf{c}, \mathbf{s}\rangle = s_i c_i + r$$

for some $r \in \mathbb{Z}_q$. This naturally leads to two cases that we now consider one by one.

$\|r\|_\infty \geq \frac{1}{2}\|\mathbf{s}\|_\infty$: Now, c_i have to be ± 1 for $s_i c_i + r < \frac{1}{2}\|s_i\|_\infty$ to ever be true. This implies

$$\mathsf{Pr}_{c \leftarrow D^d}\left[\|\langle \mathbf{c}, \mathbf{s}\rangle\|_\infty < \frac{1}{2}\|s\|_\infty \;\middle|\; \|r\|_\infty \geq \frac{1}{2}\|\mathbf{s}\|_\infty\right] \leq \mathsf{Pr}_{c_i \leftarrow D}\left[|c_i| = 1\right] = 1/2$$

which shows the first case.

$\|r\|_\infty < \frac{1}{2}\|\mathbf{s}\|_\infty$: We will show that

$$\|r + bs_i\|_\infty \geq \frac{1}{2}\|s_i\|_\infty \tag{4.1}$$

for any $b \in \{-1, 1\}$. Note that $\|s_i\|_\infty = \|\mathbf{s}\|_\infty$. This implies that

$$\mathsf{Pr}_{c \leftarrow D^d}\left[\|\langle \mathbf{c}, \mathbf{s}\rangle\|_\infty < \frac{1}{2}\|s\|_\infty \;\middle|\; \|r\|_\infty < \frac{1}{2}\|\mathbf{s}\|_\infty\right] \leq \mathsf{Pr}_{c_i \leftarrow D}\left[c_i = 0\right] = 1/2$$

which completes the proof of the lemma. Now, to prove 4.1, we first assume that $|s_i| \leq q/2$ and that $|r| < |s_i|/2$ (due to the integers being signed and the initial assumption, respectively). Therefore we can conclude that $\|r + bs_i\|_\infty$ is either equal to $|r + bs_i|$ or $|r + bs_i \pm q|$. In the first case it becomes obvious that we have $|r + bs_i| \geq |bs_i| - |r| > s_i/2$ (since $\|r\|_\infty < \frac{1}{2}\|\mathbf{s}\|$). In the latter case, assume for the sake of contradiction that $u = r + bs_i \pm q$ where $u < |s_i|/2$, then

$$q = |\pm q| = |r + bs_i - u| \leq |r| + |bs_i| + |u| < |s_i|/2 + |s_i| + |s_i|/2 = 2|s_i| \leq q$$

□

This proof follows [LNS20, Lemma A.1] closely, and is in fact a special case of this proof for $y = 0$. This particular lemma (Lemma 13) also holds true for the different distribution $D = \mathbb{B}$ [BL17] (i.e. D is sampled from $\{0,1\}^{\kappa \times d}$), but for $D_{-1,0,1}^{\kappa \times d}$ it allows for the infinity norm to be less, than when $D = \mathbb{B}$, where $\mathbb{B}$ is a uniform binary distribution. This lemma is important, as it essentially allows us to prove that the result of $\mathbf{Cs}$ is bounded, rather than prove directly that $\mathbf{s}$ is bounded.

So now, combining Corollary 1 which tells us that $\mathbf{y} = \mathbf{Cs}$ is an approximation of $\mathbf{s} \in \mathbb{F}_q^d$ with a slack of $\lambda = 9.75\sqrt{d}\|s\|_\infty$ and Lemma 13, we just need to prove smallness of $\mathbf{y}$ instead of $\mathbf{s}$, where $\mathbf{y} \in \mathbb{F}_q^\kappa$. For this there are to main strategies. Either we use *rejection sampling* or we use *binary decomposition*. According to [BN19], if the goal is to have as little slack as possible (thus allowing for a more adaptable proof of smallness as we can use more values or larger values before the slack causes the computation to wraparound modulo q), the prefered way is by hiding the result of the linear combinations (i.e. compute $\mathbf{y} = \mathbf{Cs}$) and then prove exact smallness of these by providing a bit decomposition, as this allows for a significantly better *slack*, compared to using rejection sampling.

When using rejection sampling, the prover will sample random values $x_1, x_2, \dots$ and add these to the product $\mathbf{y} = \mathbf{Cs}$ to mask the resulting values which are all supposed to be small. The prover then sends these masked values to the verifier who can simply check that they are *small enough* (i.e. below some predetermined bound). The *rejection* part comes in when one of the values from the prover doesn't get masked completely. In this case, the prover will re-sample random values $x_1, x_2, \dots$ and try again. The hope is that this rejection step won't happen too often. Additionally, the prover samples more random values than needed to hide $\mathbf{y}$ and the verifier will then ask the prover to open some of the values $x_1, x_2, \dots$ to show that they are in fact as small as the values of $\mathbf{y}$ are supposed to be. This strategy intuitively increases the slack, as the honest prover will have to add large enough values to hide the elements of $\mathbf{y}$ and since the elements $x_1, x_2, \dots$ have to be sampled from a larger space than the expected values of $\mathbf{y}$, to avoid having them be too small too often and thus cause a rejection, this results in adding values that are potentially significantly larger than the values in $\mathbf{y}$ of an honest prover [BN19]. The other approach keeps the values of $\mathbf{y} = \mathbf{Cs}$ hidden and instead have the prover commit to the bit decompositions of each element, show that these hidden values are actually bits and that they are correct bit decompositions. If the bit decompositions only consist of some predetermined number of bits, then the verifier knows that all the values of $\mathbf{y}$ are below the correct bound. This adds hardly no additional slack, but it has the additional cost of having to prove the bit decompositions are correct. So, while proving exact ranges using bit decompositions yield a better slack, it comes with slightly more computation as well as communication. As our focus is to be able to prove that as many decomposed values don't overflow when squared and summed together as possible, we consider the approach of proving exact ranges using bit decompositions. This allows us to either have a larger batch size or use larger values. If instead one has a very low batch size, the first approach could be considered for an overall cheaper protocol.

Lastly, for this to be a zero-knowledge proof of smallness, we still need to ensure that the inner product of $\mathbf{C}$ and $\mathbf{s}$ leaks no information regarding the secret vector $\mathbf{s}$ apart from the bound on each value. We have shown above that if $\|\mathbf{s}\|_\infty \leq \beta$, then since $\mathbf{C} \in \mathcal{D}_{-1,0,1}^{\kappa \times d}$, it holds that $\|\mathbf{Cs}\|_\infty \leq 9.75 \cdot \sqrt{d}\beta$ with overwhelming probability, thus we resolve the problem of leakage by finding the first r such that $9.75 \cdot \sqrt{d}\beta < 2^r$. Let $\mathbf{y} = \mathbf{Cs}$, then it suffices for the prover to show that $y_i \in [-2^r, 2^r - 1]$ for all $y_i \in \mathbf{y}$. This can be done by having the prover provide $\kappa(r+1)$ bits $x_0^i, \dots, x_r^i$ and then show that

$y_i = \sum_{j=0}^{r} 2^j x_j^i$ for each element of $\mathbf{y}$, as well as prove that each x_j^i is a bit. We recap in the following Theorem 16.

Theorem 16. *Let $k = 9.75 \cdot \sqrt{d}$ and q be such that $(q-1)/2 > 4k\beta$. Let r be the first value such that $k\beta \leq 2^r - 1$. Let λ be the slack and let $\kappa = 128$ (i.e. the dimension of the output of the multiplication with C). Then the above approach is an approximate proof-of-smallness for the ℓ_∞ norm, with slack $\lambda = 9.75\sqrt{d} + 2$*

Proof. An honest prover will have $\|\mathbf{s}\|_\infty \leq \beta/\lambda$, thus by Corollary 1 as well as the union bound[2], we have that $\|\mathbf{y}\|_\infty \leq k\|\mathbf{s}\|_\infty \leq 9.75\sqrt{d}\|\mathbf{s}\|_\infty \leq k\beta/\lambda \leq \beta/2$ except with probability $2^7 \cdot 2^{-141} = 2^{-134}$. As such, an honest prover will only not be able to represent its values by $r+1$ bits, with negligible probability.

If $\|\mathbf{y}\|_\infty \geq 2(k\beta + 2)$, then by Lemma 13 $\mathsf{Pr}[\|\mathbf{s}\|_\infty < k\beta + 2] \leq 2^{-\kappa}$. Assume for contradiction that $\|\mathbf{s}\|_\infty \geq k\beta + 2$, then there exist an index i such that either (1) $\mathbf{s}[i] \in [-(q-1)/2, -k\beta - 2]$ or (2) $\mathbf{s}[i] \in [k\beta + 2, (q-1)/2]$. In case (1), let $v_i = \mathbf{s} + k\beta$, then $v_i \in [-(q-1)/2 + k\beta, -2]$, which cannot be expressed as the sum $\sum_{j=0}^{r} 2^r x_j^i$. In case (2), $\mathbf{s}[i] \in [k\beta + 2, (q-1)/2]$, thus $v_i \in [2k\beta + 2, (q-1)/2] \cup [-(q-1)/2, -(q-1)/2 + k\beta - 1]$. This however, cannot either be represented by the sum $\sum_{j=0}^{r} 2^r x_j^i$, as $2^{r+1} - 1 = 2k\beta + 1 < 2k\beta + 2$ as well as $[-(q-1)/2, -(q-1)/2 + k\beta - 1]$ being inexpressible due to the bound on q.

Finally, to argue that the proof satisfies zero-knowledge, notice that the only thing sent by the prover is the commitment to the bits $(x_0^i, \ldots, x_r^i)_{i<\kappa}$ which are all hiding by the properties of the commitment scheme. Thus, the verifier only learns that each y_i can be represented by $r+1$ bits, but nothing else.

□

From Theorem 16 we can conclude that the overall slack of the proof is $\lambda = 9.75\sqrt{d}+2$ where d is the dimension of the secret vector. This brings down the slack significantly, compared to previous state-of-the-art using rejection sampling [GHL22] or when using exact range proofs [BN20].

4.3.2 Exact Range Proof

We now show how to use the results from the previous section to build an exact range proof, rather than an approximate one. We give the protocol in Figure 4.1 and then explain it step-by-step.

We now explain select steps of the protocol.

1. The prover has m values $\mathbf{x} = x_1, \ldots, x_m \in \mathbb{F}$ of which the verifier has commitments to. If the prover is honest, $\forall i \; x_i < \frac{\sqrt{q-1}}{78 \cdot \sqrt{m}+16}$For each value x_i, the prover wishes to show that $a_i \leq x_i \leq b_i$. The values a_i, b_i are public.

[2] Since the bound on the infinity norm comes from a bound on the ℓ_2 norm based on an inner product, we must apply this to each entrance when using the matric C

Protocol $\Pi_{\mathbf{four-squares}}$

1. $\mathcal{P}$ and $\mathcal{V}$ have m values $([x_1], \ldots, [x_m])$ and m bounds $\{(a_i, b_i)\}_{i\in[m]}$. $\mathcal{P}$ wants to show that for $\forall i : a_i \leq x_i \leq b_i$.

2. For each x_i for $i \in [m]$, both parties compute $[l_i] = [x_i] - a_i$ and $[u_i] = b_i - [x_i]$ and the prover then finds four non-negative integers $\alpha_i, \beta_i, \gamma_i, \delta_i$ such that $\alpha_i^2 + \beta_i^2 + \gamma_i^2 + \delta_i^2 = l_i \cdot u_i$. $\mathcal{P}$ commits to $\{\alpha_i, \beta_i, \gamma_i, \delta_i\}_{i\in[m]}$, resulting in $\{[\alpha_i], [\beta_i], [\gamma_i], [\delta_i]\}_{i\in[m]}$.

3. $\mathcal{P}$ commits to $\{\alpha_i^2, \beta_i^2, \gamma_i^2, \delta_i^2\}_{i\in[m]}$. $\mathcal{P}$ then shows that

   - $[\alpha_i^2] = [\alpha_i] \cdot [\alpha_i]$
   - $[\beta_i^2] = [\beta_i] \cdot [\beta_i]$
   - $[\gamma_i^2] = [\gamma_i] \cdot [\gamma_i]$
   - $[\delta_i^2] = [\delta_i] \cdot [\delta_i]$

   Let $\mathbf{s} = ([\alpha_1], [\beta_1], [\gamma_1], [\delta_1], \ldots, [\alpha_m], [\beta_m], [\gamma_m], [\delta_m])$.

4. $\mathcal{P}$ and $\mathcal{V}$ runs $\mathsf{CheckZero}(\{[l_i] \cdot [u_i] - [\alpha_i^2] + [\beta_i^2] + [\gamma_i^2] + [\delta_i^2]\}_{i\in[m]})$

5. $\mathcal{V}$ samples κ vectors $\mathbf{c_i} \leftarrow D^{4m}_{-1,0,1}$ for $i \in [\kappa]$ and sends these to $\mathcal{P}$.

6. $\mathcal{P}$ and $\mathcal{V}$ compute $[u_i] = \langle \mathbf{c_i}, \mathbf{s} \rangle$. $\mathcal{P}$ then computes bit decompositions $\{(u_{i,j})_{j\in[d/2]}\}_{i\in[\kappa]}$ of each $[u_i]$ using $d = \frac{\sqrt{q-1}}{(78\cdot\sqrt{m}+16)}$ and Lemma 13, where q is the order of the field. $\mathcal{P}$ commits to the bit decompositions so both parties get : $\{([u_{i,j}])_{j\in[d]}\}_{i\in[\kappa]}$. Both parties then run $\mathsf{CheckZero}(\{[u_i] - \sum_{j=0}^{d} 2^j \cdot [u_{i,j}]\}_{i<\kappa})$. If $\mathsf{CheckZero}$ succeeds, output accept, otherwise reject.

Figure 4.1: A protocol for showing exact range proofs based on the four-squares technique as well as an approximate range proof for each of the four squares.

2. For each x_i, both parties compute compute $l_i = x_i - a_i$ and $u_i = b_i - x_i$ and the prover then finds four non-negative integers $\alpha_i, \beta_i, \gamma_i, \delta_i$ such that $\alpha_i^2+\beta_i^2+\gamma_i^2+\delta_i^2 = l_i \cdot u_i$. The prover also prove that the multiplication of l_i and u_i is done correctly.

3. The prover must then show equality between the values $\alpha_i^2, \beta_i^2, \gamma_i^2, \delta_i^2$ and the product $l_i \cdot u_i$. The prover also has to show that each of the α's, β's, γ's and δ's are squared correctly.

4. Finally, the prover has to show that the squared values still fit within the field without overflowing and that the addition of the four squares also fit. This is done by the prover computing an approximate range proof showing that the ℓ_∞ of $\|\mathbf{w} = [\alpha_i, \beta_i, \gamma_i, \delta_i] \in \mathbb{F}^{m \cdot 4}\|_\infty < \sqrt{(q-1)/(2 \cdot 4)}$.

Lemma 14. *If* $\beta = \|\mathbf{s}\|_\infty < \frac{\sqrt{q-1}}{(78 \cdot \sqrt{m}+16)}$ *, then protocol* $\Pi_{\mathbf{four-squares}}$ *(Figure 4.1) proofs whether or not* $\forall i\ a_i \leq x_i \leq b_i$.

Proof. The approximate proof of step 4 shows that $\forall i\ w_i < \sqrt{(q-1)/(2 \cdot 4)}$, which implies that step 3 works over the integers, as no overflow can take place. To accurately prove that the above is true, we need to show that $\|\mathbf{s}\|_\infty < \sqrt{(q-1)/2 \cdot 4}$. For this to be true, the prover must in fact have all the values be smaller than $\sqrt{(q-1)/2 \cdot 4/\lambda}$ where the slack $\lambda = 9.75\sqrt{4m} + 2$. Thus, step 4 implies that the values input by the prover are bounded by

$$\frac{\sqrt{(q-1)/(2 \cdot 4)}}{(9.75 \cdot \sqrt{4m}+2)} \approx \frac{\sqrt{q-1}}{(78 \cdot \sqrt{m}+16)}$$

Which is exactly the bound in the statement of Lemma 14. This means that the bound is small enough that we can use the ℓ_∞ approximate proof from Section 4.3 to show that the decomposed elements won't overflow when squared nor when added together. This ensures that both operations work over the integers.

Zero-knowledge of the protocol is guaranteed from the hiding property of the commitment scheme as well as underlying approximation scheme. □

This concludes our range proof based on square decomposition. It achieves a significantly better slack (less of it) compared to previous work, thus allowing the prover to either prove the range of more values or higher valued values. This protocol works best for smaller numbers of range proofs at a time due to the negative impact of a higher number on the slack.

4.4 Exact Proofs of Lesser Values

This way of proving ranges falls into the n-ary decomposition, rather than the square decomposition that we saw in Sections 4.2 and 4.3.

The naïve way of proving that $x \in [0, 2^k)$ for some $x \in \mathbb{F}_p$ in this paradigm, is for the prover to provide k field elements $(x_0 \ldots, x_{k-1}) \in \mathbb{F}_p \times \cdots \times \mathbb{F}_p$, show that for each $i \in [k]$, $x_i \in \{0, 1\}$ and that $x = \sum_{i=0}^{k} 2^i \cdot x_i$, i.e., that each x_i is a bit and they are a correct binary decomposition of x. This requires committing to k field elements from $\mathbb{F}_p$ in addition to performing k multiplications to confirm each element is a bit. Using `edaBits` allows the prover to instead commit to k $x_i \in \mathbb{F}_2$. Instead, we propose dividing the element x into *chunks* of m bit-elements x_i instead, thus only requiring the prover to make k/m commitments, rather than k. The prover and verifier then check that $x = \sum_{i=0}^{k/m} 2^{i \cdot m} x_i$. This leaves the prover with the task of proving that each chunk x_i is in $[0, 2^m - 1]$. As before, this can be done by proving range of this particular field element by bit-decomposing it, but we instead propose two other methods.

Now, before we present the first method, note that we can consider the linearly-homomorphic MACs originating from VOLE as polynoimials. If we take a commitment $(x, \mathsf{M}[x])$ held by the prover and $\mathsf{K}[x]$ held by the verifier, then we can view these instead as a polynomial $f(y) = \mathsf{M}[x] - x \cdot y$. Now $\mathsf{K}[x] = f(\Delta) = \mathsf{M}[x] - \Delta \cdot x$. If we then multiply the polynomial defined above with another $f'(y) = \mathsf{M}[x'] - y \cdot x'$:

$$f \cdot f'(y) = \mathsf{M}[x]\mathsf{M}[x'] + (x\mathsf{M}[x'] + x'\mathsf{M}[x])y + xx'y^2$$

we see that the final term contains $xx' \cdot y^2$ which is the only term of degree 2. This however, is so far locally computable. The prover then define polynomial $q(y) = \mathsf{M}[z] + z \cdot y$ (i.e. commits to z) where the prover claims that $z = x \cdot x'$. Now, to show that this is true, the prover can send over the polynomial

$$y \cdot q(y) - f \cdot f'(y)$$

where $y \cdot q(y) = y\mathsf{M}[z] + zy^2$. Thus, if $z = x \cdot x'$, the term of degree 2 disappears. On the verifier side, the verifier can compute $\mathsf{K}[x]\mathsf{K}[x'] - \Delta\mathsf{K}[z]$, which again removes the term of the highest degree, like on the prover side. Rather than stop at degree 2 however, this idea generalizes to degree d, provided that the prover can provide a masking polynomial of degree $d - 1$ to hide the remaining terms. This idea is a slight simplifcation of QuickSilver [YSWW21a], since it avoids having to compute polynomials in their degree-separated form. We suspect this may be faster, if the input polynomials are simple.

Now, to explain the first method of checking the ranges, we define polynomial $f(X) = \prod_{i=0}^{2^m-1} X - i$. If $x_i \in [0, 2^m - 1]$, then $f(x_i) = 0$. Now, if $x \in [0, 2^k]$, then it holds true for each chunk $i \in [0, k/m]$ $x_i \in \mathbb{F}_p$ that $f(x_i) = 0$. To this end, we can evaluate each polynomial using the above idea. Lastly, if the used commitment scheme is additively homomorphic, we can assume w.l.o.g. that the range is always $[0, 2^k)$, as we can easily convert between this and any other range $[a, b]$. Finally, since we will always be checking

Protocol for computing range proofs

1. $\mathcal{P}$ and $\mathcal{V}$ have t values $([x^0], \ldots, [x^{t-1}])$ and $\mathcal{P}$ wants to show that $\forall i : x^i \in [0, 2^k)$.
2. For each $i \in [t]$: $\mathcal{P}$ computes the n-ary decomposition $(x^i_0, \ldots, x^i_{(k/n)-1}$ so that for each $j \in [(k/n) - 1]$: $x^i_j \in [0, 2^n)$ and $x^i = \prod_{j=0}^{k/n} 2^{j \cdot n} x_j$. $\mathcal{P}$ commits to each of the elements in the n-ary decompositions.
3. Define the polynomial $f(X) = \prod_{i=0}^{2^n-1} X - i$. Now, $\mathcal{P}$ proves satisfiability of $f(x^i_j$ for each $i \in [t]$ and $j \in [n]$.
4. If the evaluation passes, then each polynomial evaluated to 0 and $\mathcal{V}$ accepts. Otherwise, $\mathcal{V}$ rejects.

Figure 4.2: Protocol for computing range proofs using QuickSilver.

that the multiplication is 0, further savings can be made, such as knowing that the solution polynomial $q(y)$ should always have 0 in its highest degree.

The second method we consider, is using a disjunction proof from Mac'n'Cheese [BMRS21a]. If we again consider the polynomial $fX) = \prod_{i=0}^{2^m-1} X - i$, notice then that exactly one of the products will be 0, whereas the others will be non-zero. Therefore, it can be stated as a disjunction proof where the prover shows that exactly one of the products is 0.

Rather than pick a specific way of showing the ranges now, we will resort to write that we must prove satisfiability of the polynomials for now.

We present the full protocol in Figure 4.2

We suspect that the main cost of this protocol comes in form of communication when having to commit to the n-ary decomposition as well as the polynomial evaluation. Both parts can be tuned in terms of efficiency, as decomposing into larger elements decreases the number of commitments, at the cost of increasing the degree of the polynomials that must be evaluated, thus worsening the efficiency of the polynomial evaluation. We suspect there is a point where this is most efficient, but testing is required to find this point.

4.5 Idea based on Polynomials

In this section we present a protocol that is based on binary decomposition. Given commitments $[a_1], \ldots, [a_n]$ over a large finite field $\mathbb{F}_q$, we want to prove that all $a_i \in [0, 2^m)$ for some $m < \log_2(|\mathbb{F}_q|)$. That is, they can be represented as $a_i = \sum_{j=1}^{m} 2^{j-1} \cdot b_{i,j}$ with all $b_{i,j} \in \{0, 1\}$.

Idea Consider polynomials $p_1, \dots, p_m, q_1, \dots, q_m \in \mathbb{F}_q[X]$ such that $p_j(i) = b_{i,j}$ and $q_j(i) = 1 - b_{i,j} = 1 - p_j(i)$ for $i \in [n], j \in [m]$. Then we have the following equalities:

$$p_j(i) \cdot q_j(i) = 0 \text{ for } i \in [n], j \in [m] \tag{4.2}$$

$$p_j(i) + q_j(i) = 1 \text{ for } i \in [n], j \in [m] \tag{4.3}$$

$$a_i = \sum_{j=1}^{m} 2^{j-1} \cdot p_j(i) \text{ for } i \in [n] \tag{4.4}$$

If the prover can commit to the p_j and q_j and show that the three equalities hold, then it has proven $a_i \in [0, 2^m)$ for all $i \in [n]$: From (4.2), we know that at least one of $p_j(i)$ and $q_j(i)$ is 0 since $\mathbb{F}_q$ is a field. From (4.3), we get, that only one of the $p_j(i)$ and $q_j(i)$ can be 0 and the other must be 1. Hence, $p_j(i) \in \{0, 1\}$ and $q_j(i) = 1 - p_j(i)$. Finally, (4.4) shows that $(p_1(i), \dots, p_m(i))$ is the bit decomposition of a_i, and thus a_i can have a value of at most $2^m - 1$.

4.6 Evaluation

We have implemented the square decomposition technique of Section 4.3.2 in Rust using the Swanky Library [Gal19] and compared this to our own implementation of range proofs using the conversion check described in [BBMH+21b] (also implemented in Rust). Note that both of these implementations are rudimentary and both has room for optimizations.

In Table 4.1 we compare the time in microseconds on average per range proof while varying the bit sizes. Due to the nature of the approximate range proof, we must either keep the values low or the batch sizes low, this is however already a discussed compromise. We see that the protocol based on square decomposition (Figure 4.1) far outperforms the protocol described in [BBMH+21b] when running on low values (and few values). It is worth noting that this is expected, as [BBMH+21b] is designed to work on much larger batches as well as large values. It is however our only other alternative to the complete naïve implementation.

Range	2^{12}	2^{13}	2^{14}
Total time per proof (in μs) (A2B)	201.97	212.19	242.52
Total time per proof (in μs) (Our)	26.855	27.27	26.43

Table 4.1: Timing our square decomposition proof versus A2B [BBMH+21b] when performing 1000 range proofs using the ranges $2^{12}, 2^{13}, 2^{14}$.

In Table 4.2 we show that the running time of our square decomposition protocol is roughly constant, regardless of the number of range proofs.

In Table 4.3 the story is the same as for Table 4.1. Our protocol uses far less communication (which is expected), as the square decomposition protocol only requires

Proofs	1000	1500	2000
Total time per proof (in μs) (Our)	26.855	27.27	26.43

Table 4.2: Timing our square decomposition proof when performing $1000, 1500$ or 2000 range proofs using the range 2^{13}.

sending over 4 field elements (in addition to the approximate range proof), whereas the protocol that utilizes the conversion check of [BBMH+21b] must send over 5 field elements from $\mathbb{F}_p$ in addition to a lot of field elements from $\mathbb{F}_2$. It is however worth noting that it is infeasible to use our protocol based on square decomposition on the number of values that best suit [BBMH+21b], as the approximate range proof would be insecure to run on basically any value at this point.

Range	2^{12}	2^{13}	2^{14}
Total communication (in KB) (A2B)	225.25	236.00	246.75
Total communication (in KB) (Our)	83.22	83.22	83.22

Table 4.3: Measuring the communication of our square decomposition proof versus A2B [BBMH+21b] when performing 1000 range proofs using the ranges $2^{12}, 2^{13}, 2^{14}$.

Finally, in Table 4.4 we see that both protocols scale basically linearly in the communication.

Number of proofs	1000	1500	2000
Total communication (in KB) (A2B)	236.00	349.77	464.13
Total communication (in KB) (Our)	83.22	118.38	153.53

Table 4.4: Measuring the communication of our square decomposition proof versus A2B [BBMH+21b] when performing $1000, 1500, 2000$ range proofs using on the range 2^{13}.

For context, if we instead run [BBMH+21b] using insecure parameters (i.e. at its most optimal, but with a number of proofs that would be insecure), the communication drops to $291.53KB$ for 2000 range proofs on the range 2^{13}. While this is still slightly more, note that it scales far beyond what we can do with square decomposition.

Finally, we have also implemented a prototype of our protocol using n-ary decomposition of Section 4.4. This version only contains the computation of the polynomials and then the variant of QuickSilver [YSWW21a] that we describe in Section 4.4. Some things are missing from our QuickSilver implementation, such as the masking of the polynomial that is sent from the prover to the verifier (and in turn the protocol for creating this mask). This does mean that the following numbers are very rough estimates (and all lower than what they would be in the final protocol), but the purpose is to give a general gist of the effectiveness of the protocol.

In Table **??** we list the communication required by our prototype to prove between 1000 and 2000 proofs, while decomposing the range 2^{24} into smaller chunks of size either

Number of proofs	1000	1500	2000
Total communication (in KB, 2^4	47.0	70.56	94.0
Total communication (in KB, 2^6)	31.75	47.88	63.5

Table 4.5: Measuring the communication of one of our n-ary decomposition protocols when we show exact ranges where we use polynomials of degree 2^4 or 2^6. We run either $1000, 1500$ or 2000 proofs on the range $[0, 2^{24})$.

2^4 or 2^6 (i.e. $m = 4$ or $m = 6$). We see that more communication is required for the smaller size, as this means sending over more elements, however as we see in Table 4.6, this comes at a cost. It is significantly more expensive to compute the polynomials when $m = 6$.

Number of proofs	1000	1500	2000
Total time per proof (in μs, 2^4	9.37	10.93	11.02
Total time per proof (in μs, 2^6)	65.43	71.6	71.24

Table 4.6: Measuring the time of one of our n-ary decomposition protocols when we show exact ranges where we use polynomials of degree 2^4 or 2^6. We run either $1000, 1500$ or 2000 proofs on the range $[0, 2^{24})$.

While all of the numbers for our n-ary decomposition protocol are rough estimates, they show that we can pick m (i.e. the bit size of the values in the decomposition), to alter either the communication or the computation required for the proofs. What remains to be implemented, is the generation of the polynomial mask, which adds the communication of a few extra field elements to the cost. Despite this, these initial benchmarks show that this protocol has high potential.

Comparing the n-ary decomposition protocol to the square decomposition protocol, we see that the latter has an almost constant running time per proof, regardless of the size of the range or of the number of proofs, whereas the former increases in cost based on the number of proofs or the size of the ranges.

Chapter 5

Pecorino: More Efficient Zero-Knowledge for RAM Programs

5.1 Introduction

Zero-knowledge (ZK) proofs allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that it knows some input w such that $P(w) = 1$ for program P, with $\mathcal{V}$ learning nothing else about w besides that it satisfies P. There has been a huge amount of work in recent years improving the concrete efficiency of ZK protocols, both in the non-interactive and interactive settings. However, most such protocols require that P be represented as a (boolean or arithmetic) circuit or constraint system, which differs from the standard RAM-model of computation many programs are written in.

To address this, Ben-Sasson et al. [BCG+13] introduced an approach for compiling RAM programs to circuits or constraint systems using the following framework:

1. $\mathcal{P}$ commits to a list L of memory operations `(addr, time, 'read' | 'write', value)` during the course of the program execution.

2. $\mathcal{P}$ sorts this list by `(addr, time)`, producing list L', commits to L', and proves to $\mathcal{V}$ that L' is a valid sorting of L.

3. $\mathcal{P}$ and $\mathcal{V}$ iterate through L' to check that the memory operations are consistent, i.e., that the value read after a write matched the read value.

The most expensive part of this process is usually the sort in Step 2, which inherits a $O(\log T)$ overhead for T accesses to memory due to a permutation network alongside an assertion that the permutation was correctly sorted. Recently, Franzese et al. [FKL+21] introduced a new approach—henceforth referred to as the FKLOWW protocol—which replaces the permutation network with a linear *permutation check*. This approach allows for RAM computations over $\mathbb{F}_2$ which scales linearly in the word size with the number of RAM accesses of the program.

In this work, we build off of the FKLOWW protocol, introducing a ZK protocol that (1) scales linearly in both the word size *and* the underlying circuit size, (2) works over arbitrary fields, and (3) has improved performance both in terms of communication and computation.

5.1.1 The FKLOWW Protocol

Our starting point is the FKLOWW paper discussed above. In this approach—and similar to the RAM framework of Ben-Sasson et al. [BCG+13]—both parties maintain

a (committed) list L of all memory accesses, alongside a "timestamp" counter `time` initialized to zero. We use the notation $[\cdot]$ to denote a committed value.

For a read or write operation `op` on a committed address [`addr`], $\mathcal{P}$ generates a commitment to the value [`value`] and the parties store ([`addr`], [`time`], [`op`], [`value`]) in L and increment `time`. Once all RAM operations have completed, the parties proceed to a *check phase*: $\mathcal{P}$ (locally) sorts L by its first then second entry and generates commitments for the values of this sorted list—we call the list of sorted tagged values L'. The parties then check (in ZK) that L' is indeed a correct sorting. Finally, the parties run a polynomial equality check to prove that L' contains the same elements as L, hence proving that L' is a sorting of L.

The key insight of the FKLOWW protocol is to replace the sorting network used to prove that L' is a sorting of L—which requires $O(\log T)$ overhead—with a polynomial equality check—which requires only $O(1)$ overhead. This polynomial equality check works as follows: Given $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, let $A(X) = \prod_i (X - a_i)$ and $B(X) = \prod_i (X - b_i)$. If the lists a and b are permutations of each other, then the polynomials A and B are equal. We can thus efficiently check whether $A \stackrel{?}{=} B$ by having the verifier send the prover a challenge, $e \in \mathbb{F}$, and asserting that $A(e) = B(e)$.

While the FKLOWW protocol is significantly more performant than prior work, it suffers from two main drawbacks: (1) it requires $\mathcal{P}$ to prove that its sorted list L' is correctly sorted by proving that entry i is less than or equal to entry $i+1$, which, while linear in the word size, requires $O(\log T)$ AND gates, thus making the computation superlinear in the underlying circuit complexity, and (2) due to these comparison operations, it only works over boolean extension fields.

5.1.2 Our Approach

In this work, we build off of the FKLOWW protocol to construct a ZK-RAM protocol with linear overhead in both the word size *and* circuit complexity[1] that also works over any field.

The key idea of our approach is the following:

1. Rather than *sorting* memory operations, we *assert* that a set of read operations equals a set of write operations. This means that instead of combining read and write operations in a single list, we maintain *two* lists: one for read operations and one for write operations.

2. We ensure that the read and write operations respect time by using the order of commitments with respect to verifier challenges.

The read and write lists consist of triples consisting of (`addr`, `value`, `challenge`). $\mathcal{P}$ and $\mathcal{V}$ maintain two lists: a list of reads $\mathcal{R}$ and a list of writes $\mathcal{W}$. Initially, $\mathcal{W}$ contains

[1] Given a fixed field $\mathbb{F}$. In particular, neither our approach nor the FKLOWW protocol are truly linear in either the memory size N or the number of memory accesses T due to achieving a security bound of $O(N+T)/|\mathbb{F}|$. That is, as N and T grow, the chosen field $\mathbb{F}$ must grow as well.

one entry for each value in RAM (that is, the initial size of $\mathcal{W}$ equals the size of RAM), where each `challenge` is set to 0. Values are initialized as desired by the parties (to configure the initial RAM values). $\mathcal{R}$ is initialized as empty.

The prover maintains a set $\mathcal{W}^*$ of "hot" writes (writes which have not been read yet). Initially, this set is equal to $\mathcal{W}$.

Rather than exposing distinct read and write operations, we instead (conceptually) expose only a single operation:

```
fn ReadAndWriteBack(
    addr,
    compute_new_value: fn(old_value) -> new_value
) -> (old_value, new_value)
```

where `compute_new_value` is a function/closure which determines what value should be written back as part of this operation. This closure is not allowed to perform any memory operations on `addr`, but it *is* allowed to perform memory operations on other addresses.

`ReadAndWriteBack`$([\texttt{addr}], f)$ is implemented as follows.

1. $\mathcal{P}$ removes the entry containing `addr` from $\mathcal{W}^*$, and determines the value `value` and challenge `challenge` stored in that entry.

2. $\mathcal{P}$ commits to $[\texttt{value}]$ and $[\texttt{challenge}]$.

3. Both parties add $([\texttt{addr}], [\texttt{value}], [\texttt{challenge}])$ to $\mathcal{R}$.

4. The parties compute $[\texttt{value}'] \leftarrow f([\texttt{value}])$.

5. $\mathcal{V}$ sends $\mathcal{P}$ a random field element $\texttt{challenge}'$.

6. The parties add $([\texttt{addr}], [\texttt{value}'], \texttt{challenge}')$ to $\mathcal{W}$, and $\mathcal{P}$ adds $(\texttt{addr}, \texttt{value}', \texttt{challenge}')$ to $\mathcal{W}^*$.

Once all memory operations have been completed, $\mathcal{P}$ commits to all the entries in $\mathcal{W}^*$ so that both parties can add $\mathcal{W}^*$ to $\mathcal{R}$. (Note that the size of W^* is always the size of memory.) Finally, the parties use the same permutation check as Franzese et al. [FKL+21] (originally due to Neff [Nef01]) to assert that $\mathcal{R}$ and $\mathcal{W}$ are permutations of each other.

For the sake of exposition—and to more easily compare with the "standard" read-write RAM interface of other works—we simplify $\mathsf{ReadAndWriteBack}$ by applying a restriction on the function f, so that $\mathsf{ReadAndWriteBack}$ strictly results in a Read or a Write operation, while hiding this operation. This simplifies comparison with previous work, but leaves open space for more abstract protocols, which we discuss in more detail in item 5.3.2.

Security

Informally, since $\mathcal{P}$ shows that $\mathcal{R}$ and $\mathcal{W}$ are permutations of each other, and since each memory entry contains an address, it is not possible for $\mathcal{P}$ to cheat by swapping values intended for one address with values intended for another address. $\mathcal{P}$ also cannot claim that a read returned a value that was never written to that address, since that would also cause the permutation check to fail.

Now the only remaining attack would be $\mathcal{P}$ re-ordering reads (so that a later write output was given to a previous read). This is not possible since $\mathcal{P}$ needs to commit to the read's challenge *before* $\mathcal{V}$ has sent it the challenge for any subsequent writes. Thus, $\mathcal{P}$ is unable to guess the challenge of the subsequent read, so it cannot successfully move it earlier. If $\mathcal{P}$ tries to use the challenge of a previous read, then it will be neglecting the challenge of the write that it should be matched up with. As a result, the permutation assertion fails. We provide a formal argument of this claim in Theorem 18.

Reducing the Number of Rounds

The protocol as presented requires $\mathcal{V}$ to send a challenge value for every read/write operation. Unfortunately, this adds overhead when run in practice, as the protocol must pay the latency cost of communicating the challenge on each operation. While we can avoid this by using the Fiat-Shamir transformation [FS87], doing so comes at a significant security loss, particularly since our protocol has a number of rounds equal to the number of RAM accesses. We show in subsection 5.3.3 an approach that removes this per-round communication while avoiding the need for Fiat-Shamir. In particular, we present an approach that allows the protocol to proceed in *batches*, where the verifier needs to only communicate between batches, and all RAM accesses within a given batch do not require communication.

5.1.3 Efficiency

Let ρ be the statistical security parameter, T the number of memory operations, N the memory size, W the bit length of the values stored in RAM (i.e., the word size), and B the batch size (i.e., the number of RAM accesses per batch). For the fully interactive mode we have $B = 1$ and for the fully non-interactive mode we have $B = T$.

Below, we compare our approach with the FKLOWW protocol. We begin by introducing some terminology used in the comparison. Both our approach and the FKLOWW protocol are compatible with VOLE-style ZK protocols, such as Quicksilver [YSWW21a] and Mac'n'Cheese [BMRS21a]. In these protocols, *vector oblivious linear evaluation* (VOLE) [WYKW21a] is used to efficiently generate linearly homomorphic commitments $[r]$ to random values r known to the prover. These values can then be "fixed" by the prover to particular values of its choosing by sending the verifier a "fix" value f and having each party compute $[r] - f$. We thus use the terminology "Fix bits" to denote the number of bits that the prover needs to send. Likewise, "challenge bits" denotes the number of bits the verifier needs to send.

Below, we compare the cost of our approach and the FKLOWW protocol across four metrics: (1) the number of Fix bits required, (2) the number of challenge bits required, (3) the number of AND gates required, and (4) the number of multiplication gates (over some arbitrary finite field $\mathbb{F}$[2]) required. Note that the FKLOWW protocol requires no challenge bits, and our approach requires no AND gates.

Read-Only Memory

For read-only memory we have the following costs:

	Our Approach	FKLOWW
Fix **Bits**	$(\rho + W)(T + N)$	$(\rho + W)T$
Challenge Bits	$\rho \cdot T/B$	0
# AND Gates	0	$(W + 2\log N)(N + T)$
# Mult Gates	$2(T + N - 1)$	$2(T + N - 1)$

As both protocols use the same approach to checking a permutation, the cost in number of multiplication gates is the same. However, the FKLOWW protocol requires a large number of AND gates to enforce the correct sorting of the list L', which our approach avoids. In addition, for cases where the number of memory operations T is smaller than the size of memory N, the FKLOWW protocol requires fewer Fix bits.

Read-Write Memory

For read-write memory, we follow the convention of Franzese et al. and assume memory is not initialized to private values (and memory is only initialized through the T memory operations).

	Our Approach	Franzese et al.
Fix **Bits**	$(\rho + W)(T + N)$	$(\rho + W + \log N + 1)T$
Challenge Bits	$\rho \cdot T/B$	0
# AND Gates	0	$T(5 + 2\log N + W + \log T)$
# Mult Gates	$N + 2(T - 1)$	$2(T - 1)$

Note that the above represents a worst-case comparison for our approach, since each memory operation is both a read *and* a write, while in FKLOWW it is a read *or* a write. As a result, for many programs, the T value should be halved.

In general, our approach works better for programs that do many accesses on a smaller amount of memory. If $T \geq N$ (the program does at least as many memory accesses as there are addresses), and those T memory operations are read-then-write-back memory operations (so that each pair of memory operations can be performed as only a single `ReadAndWriteBack` operation), then our approach performs better on every metric above (except for challenge bits).

[2] As an example, for binary computations $\mathbb{F}$ would in practice be $\mathbb{F}_{2^{40}}$.

Concrete Comparison To put concrete[3] communication numbers to the comparison, consider a RAM size of 2^{24} with a 32-bit word size and a statistical security parameter of $\rho = 40$. Ignoring the communication cost of VOLE preprocessing, for $T = 2^{26}$ memory accesses, our approach requires only 30 bytes per memory access, compared to FKLOWW's required 34 bytes per memory access.

5.1.4 Related Work

There has been a long line of work developing zero-knowledge proofs in the RAM model. Early work by Ben-Sasson et al. [BCG+13, BCTV13] and others [WSH+14] utilized a routing network to enforce consistency of memory accesses. This approach was further utilized in garbled-circuit zero-knowledge protocols [JKO13, FNO15] by Heath and Kolesnikov [HK20b], resulting in the BubbleRAM [HK20a] and BubbleCache [HYDK21] systems. However, the routing network approach inherently comes with a logarithmic overhead per memory access, resulting in relatively high communication per memory access: as an example, with a memory size of $N = 2^{24}$ and a 32-bit word size, BubbleCache requires 240 bytes of communication.

An alternative approach is to utilize oblivious RAM as the means to hiding RAM accesses [HMR15, MRS17]. However, this line of work focused on asymptotic performance and resulted in large hidden constants, making an implementation impractical.

As discussed in the Introduction, Franzese et al. [FKL+21] replaced the need for a routing network (and in turn the use of oblivious RAM), by instead utilizing a polynomial-based permutation check which then ensures consistency of memory accesses. Using this approach, they are able to achieve linear (in the word size) communication complexity and concretely-efficient communication and computation for both the prover and the verifier. More recently, de Saint Guilhem et al. [DOTV22] proposed a public-coin constant-overhead zero-knowledge protocol in the RAM model which works over fields of any characteristic, compared to only boolean in prior work. They achieve this by modifying Franzese et al.'s approach, achieving the best known result in the public-coin setting.

5.2 Preliminaries

Our protocol makes use of a $\mathcal{F}_{\mathsf{zk}}$ functionality which abstracts away functionality that allows the prover to commit to values, and for both the prover and verifier to compute over committed values. See Figure 5.1. This functionality can be instantiated using several existing protocols [WYKW21a, DIO20, YSWW21a, BBMH+21b, BBMHS22a] that all have good concrete efficiency while enabling bit packing, which our construction

[3] We stress that the communication numbers presented in this paragraph are instantiated with FKLOWW's permutation check. Our implementation of our protocol is optimized for a compute-bound setting (see section 5.5), and so we implemented a permutation check with less computation, in exchange for more communication.

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

Inputs: On input (Input, x) from $\mathcal{P}$, store x and send $[x]$ to both parties.
Constants: On input (Const, x) from both parties, store x and send $[x]$ to both parties.
Circuit evaluation: On input $(\mathsf{Compute}, C, [x_1], \ldots, [x_n])$ from both parties, compute $y \leftarrow C(x_1, \ldots, x_n)$ and send $[y]$ to both parties.
Open: On input $(\mathsf{Open}, [x])$ from both parties, send x to both parties.

Figure 5.1: Ideal functionality for stateful ZK proofs.

Functionality $\mathcal{F}_{\mathsf{zk\text{-}array}}$

Initialize: On input $(\mathsf{Initialize}, N)$ from both parties, initialize $M_1, \ldots, M_N$ to 0 and set $\mathsf{result} \leftarrow \mathsf{honest}$.
Access: On receiving $(\mathsf{Access}, [\mathsf{op}], [a], [v])$ from both parties:

1. If $a > N$, set $\mathsf{result} \leftarrow \mathsf{cheating}$.
2. If $\mathsf{op} = \mathsf{Read}$, send $[M_a]$ to both parties.
3. If $\mathsf{op} = \mathsf{Write}$, set $M_a \leftarrow v$ and send $[v]$ to both parties.

Check: On input Check from $\mathcal{V}$ proceed as follows: if $\mathcal{P}$ sends cheating then send cheating to $\mathcal{V}$, otherwise if $\mathcal{P}$ sends continue then send result to $\mathcal{V}$.

Figure 5.2: Ideal functionality for private read/write RAM access.

uses when operating over $\mathbb{F}_2$. In particular, we utilize packing as part of the permutation check when operating over $\mathbb{F}_2$ (cf. section 5.4).

The goal is to realize the $\mathcal{F}_{\mathsf{zk\text{-}array}}$ functionality presented in Figure 5.2. This functionality has three interfaces. Initialize initializes the RAM to be of size N. Access is then used to read or write to the RAM. It takes as input commitments to an operation (either 'read' or 'write'), an address, and a value. Upon a read, the committed value at the given address is returned to both parties, and upon a write, the value is written to the given address and a committed version of that value is returned to both parties. Finally, Check allows the verifier to check consistency of RAM.

Finally, in the implementation of our protocol—and in the concrete instantiation of the permutation check—we utilize VOLE-based ZK protocols. We thus provide a brief description of VOLE below. Vector oblivious linear evaluation (VOLE) is a two-party function between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$ in which $\mathcal{S}$ and $\mathcal{R}$ obtain correlated random vectors of the following form: $\mathcal{S}$ obtains two vectors $\mathbf{x} \in \mathbb{F}_V^n$ and $\mathsf{M} \in \mathbb{F}_T^n$ and $\mathcal{R}$ gets a random scalar $\Delta \in \mathbb{F}_T$ and a random vector $\mathsf{K} \in \mathbb{F}_T^n$ where $\mathbb{F}_V$ is a (not necessarily

proper) subfield of the finite field $\mathbb{F}_T$ [WYKW21a, BBMHS22a]. The VOLE function ensures that the correlation $\mathsf{M} = \Delta \cdot \mathbf{x} + \mathsf{K}$ is satisfied. In VOLE-based ZK protocols, the prover performs the role of $\mathcal{R}$ and the verifier performs the role of $\mathcal{S}$.

5.3 New Protocol for ZK Proofs in the RAM Model

In this section we present our novel zero-knowledge proof in the RAM model. We focus on two variants: In these the prover first commits to an array of values and then (1) in the read-only case, reads values from this array while hiding both the address which is read from, as well as the value read, or, (2) in the general case, reads/writes values from/to the array while hiding both the operation, address and values that are read or written.

The keystone of our constructions is a function we call ReadAndWriteBack. As touched upon in subsection 5.1.2, this function is very general, but for the sake of exposition, we restrict it slightly. We present the restricted variant here. First off, during every call to ReadAndWriteBack, we utilize three different lists, each of which is used in every call. We first describe these lists:

$\mathcal{R}$ After a value has been read from the memory, both the prover and the verifier append the handles for the address, the value, and the challenge value to this list.

$\mathcal{W}$ After a value has been written to the memory, both the prover and the verifier append the handles for the address and the value to this list, but we leave the random challenge value in plaintext. Initially, this list contains N tuples and it represents the initial memory.

$\mathcal{W}^*$ After a value has been written to the memory, the prover adds the address, the new value that was written as well as the corresponding challenge value to this list. All values are kept in plaintext. The prover removes a tuple corresponding to the provided address from this list whenever an address is read from memory. Initially, this list is equal to $\mathcal{W}$, apart from it containing the plaintext versions.

We now describe the restricted ReadAndWriteBack. The function takes as argument commitments to an address $[a]$, a value $[v']$ as well as an operation $[\mathsf{op}]$. At a high level, this function:

1. Performs a read by having the prover look up the address in plaintext in $\mathcal{W}^*$ and then re-committing to the value v and randomness value c from there. Both $\mathcal{P}$ and $\mathcal{V}$ then add $([a], [v], [c])$ to $\mathcal{R}$.

2. Both parties agree on a shared function f. Both parties then compute the function $[y] \leftarrow f([v], [v'], [\mathsf{op}])$. Lastly, $\mathcal{V}$ send a new random value c'.

3. Now both $\mathcal{P}$ and $\mathcal{V}$ write $([a], [y], c')$ to $\mathcal{W}$ and $\mathcal{P}$ writes (a, y, c') to $\mathcal{W}^*$, in preparation for the next operation.

Functionality $\mathcal{F}_{\mathsf{zk\text{-}ro\text{-}array}}$

Initialize: On input $(\mathsf{Initialize}, v_1, \ldots, v_N, T)$ from both parties, construct list $L := (v_1, \ldots, v_N)$ and set $\mathsf{result} \leftarrow \mathsf{honest}$.
Cheat: On receiving (Cheat) from $\mathcal{P}$, set $\mathsf{result} \leftarrow \mathsf{cheating}$.
Read: On receiving $(\mathsf{Read}, [a], v)$ from $\mathcal{P}$ and (Read) from $\mathcal{V}$, send $[L_a]$ to both parties.
Check: On input Check from $\mathcal{V}$ proceed as follows: if $\mathcal{P}$ sends $\mathsf{cheating}$ then send $\mathsf{cheating}$ to $\mathcal{V}$, otherwise if $\mathcal{P}$ sends $\mathsf{continue}$ then send result to $\mathcal{V}$.

Figure 5.3: Ideal functionality for private read-only RAM access.

Following this idea, we can tailor ReadAndWriteBack to either do a read or write operation exclusively, by specifying restrictions on the function f. In subsection 5.3.1 we define how to tailor ReadAndWriteBack to stricly perform read operations and in subsection 5.3.2 we extend the function to allow for both reading and writing. We elaborate on the idea of performing both operations for each call, in item 5.3.2.

5.3.1 Read-Only Memory

We begin by describing a simplified version of our full protocol. This version only allows the prover to read from the shared memory, while keeping both the address and value private.

A formal description of the functionality $\mathcal{F}_{\mathsf{zk\text{-}ro\text{-}array}}$ is given in Figure 5.3. Our protocol realizes the functionality $\mathcal{F}_{\mathsf{zk\text{-}ro\text{-}array}}$ in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model and works in three phases as described briefly below. See Figure 5.4 for the formal description.

We define the function Read from the function ReadAndWriteBack (as informally described in section 5.3) by essentially removing the operation parameter op as well as removing the function f, as we are interested in forcing $\mathcal{P}$ to write the same value back to $\mathcal{W}$ and $\mathcal{W}^*$, as was read in $\mathcal{W}^*$ in the beginning of the call. This allows $\mathcal{P}$ to read the same value several times.

Initialization The prover initializes the memory by generating the list $\mathcal{W}^*$ and generating the handles constituting $\mathcal{W}$. The prover generates N handles $([a], [v], [c])$ which represents the initial memory. The prover must also keep track of $\mathcal{W}^*$, which represents all of the values in memory which has not been read yet. Thus, initially $\mathcal{W} = \mathcal{W}^*$, except all values in the prover's private $\mathcal{W}^*$ are stored in the clear.

Read The prover looks up the address a in $\mathcal{W}^*$ and then privately commits to the value and challenge. It then removes the a triple from $\mathcal{W}^*$ and both parties append $([a], [v], [c])$ to $\mathcal{R}$. $\mathcal{V}$ sends $\mathcal{P}$ a new random value c'. Both parties add the tuple $([a], [v], c')$ to $\mathcal{W}$ and $\mathcal{P}$ adds (a, v, c') to $\mathcal{W}^*$.

Check In order for a sequence of Read operations to be correct, the value returned by each Read must match the value initially written to the given address. To check this, $\mathcal{P}$ commits to $\mathcal{W}^*$, and both parties add the committed $([a],[v],[c])$ tuples to $\mathcal{R}$. Note that the Read operations are correct if $\mathcal{W}$ is a permutation of $\mathcal{R}$; that is, the tuples $([a],[v],[c])$ being contained in both $\mathcal{W}$ and $\mathcal{R}$ ensures that $\mathcal{P}$ never read something that was not already present in the memory $\mathcal{W}$. Thus, $\mathcal{P}$ and $\mathcal{V}$ use the permutation check protocol from section 5.4 to ensure that $\mathcal{W}$ and $\mathcal{R}$ are permutations.

In the proof, as well as the protocol, we use a function denoted as Pack. This function packs multiple elements together into a single element, and we formally define this in section 5.4.

Theorem 17. *Protocol $\Pi_{\textsf{zk-ro-ram}}$ securely realizes the functionality $\mathcal{F}_{\textsf{zk-ro-array}}$ in the $\mathcal{F}_{\textsf{zk}}$-hybrid model with statistical error $(N+T+1)/|\mathbb{F}|$, where N is the size of memory and T is the number of read operations.*

Proof. We build a simulator Sim interacting with the $\mathcal{F}_{\textsf{zk-ro-array}}$ functionality as follows. Let $\mathcal{P}^*$ be the prover; Sim runs $\mathcal{P}^*$ as a subroutine and runs as follows:

- **Initialize**: For $i \in [N]$, Sim constructs (a_i, v_i, c_i) from the calls $\mathcal{P}^*$ makes to $\mathcal{F}_{\textsf{zk}}(\textsf{Const})$, storing the results in list $\mathcal{W}$. Sim initializes list $\mathcal{R} := \emptyset$ and $\mathcal{W}^* := \mathcal{W}$. Sim sends $(\textsf{Initialize}, v_1, \ldots, v_N, T)$ to $\mathcal{F}_{\textsf{zk-ro-array}}$.
- **Read**: Sim proceeds as follows:
  - Sim extracts a from its handle $[a]$.
  - Let v and c be the values $\mathcal{P}^*$ sends to $\mathcal{F}_{\textsf{zk}}(\textsf{Fix})$. Sim looks up (a, v, c) in $\mathcal{W}^*$ and removes it. If Sim cannot find the tuple in $\mathcal{W}^*$ it calls $\mathcal{F}_{\textsf{zk}}(\textsf{Cheat})$.
  - Sim sends random challenge $c' \leftarrow \mathbb{F}$ to $\mathcal{P}^*$, and adds (a, v, c) to $\mathcal{R}$, (a, v, c') to $\mathcal{W}$, and (a, v, c') to $\mathcal{W}^*$.
  - Sim sends $(\textsf{Read}, [a], v)$ to $\mathcal{F}_{\textsf{zk-ro-array}}$.
- **Check**: Sim proceeds as follows:
  - For $i \in |\mathcal{W}^*|$: Sim obtains a_i, v_i, and c_i from $\mathcal{F}_{\textsf{zk}}(\textsf{Fix})$ and adds (a_i, v_i, c_i) to $\mathcal{R}$. If (a_i, v_i, c_i) does not match the ith tuple in $\mathcal{W}^*$, Sim calls $\mathcal{F}_{\textsf{zk}}(\textsf{Cheat})$.
  - Sim sends random challenge $z \leftarrow \mathbb{F}$ to $\mathcal{P}^*$.
  - For $i \in [N]$, let $x_i \leftarrow \textsf{Pack}(\mathcal{R}[i])$ and $y_i \leftarrow \textsf{Pack}(\mathcal{W}[i])$. Sim obtains the values $x_i - z$ sent by $\mathcal{P}^*$ to $\mathcal{F}_{\textsf{zk}}(\textsf{Compute})$. If $\textsf{PolyCheck}(x_1 - z, \ldots, x_N - z) \neq 1$, Sim sends cheating to $\mathcal{F}_{\textsf{zk-array}}$ and aborts, outputting whatever $\mathcal{P}^*$ outputs. Otherwise, Sim sends continue to $\mathcal{F}_{\textsf{zk-array}}$ and halts, outputting whatever $\mathcal{P}^*$ outputs.

Protocol $\Pi_{\mathsf{zk\text{-}ro\text{-}ram}}$

Parameters:

- N: Size of memory.
- $\mathbb{F}$: Field to use in protocol.

Common circuits:

- $\mathsf{PolyCheck}$: Arithmetic circuit defined as follows:
  $\mathsf{PolyCheck}(x_1, \ldots, x_M, y_1, \ldots, y_M) = 1$ if and only if $\prod x_i = \prod y_i$.

$\mathsf{Initialize}(v_1, \ldots, v_N)$:

1. For $i \in [N]$: $\mathcal{P}$ and $\mathcal{V}$ compute $([a], [c]) \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Const}, (i, 0)$ and $[v_i] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, v_i)$, storing the result in list W.
2. $\mathcal{P}$ and $\mathcal{V}$ set list $\mathcal{R} := \emptyset$. $\mathcal{P}$ sets list $\mathcal{W}^* := \mathcal{W}$.

$\mathsf{Read}([a])$:

1. $\mathcal{P}$ finds entry $(a, v, c) \in \mathcal{W}^*$ and removes it.
2. $\mathcal{P}$ computes $[v] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, v)$ and $[c] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, c)$.
3. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [v], [c])$ to $\mathcal{R}$.
4. $\mathcal{V}$ sends random challenge $c' \leftarrow \mathbb{F}$ to $\mathcal{P}$.
5. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [v], c')$ to $\mathcal{W}$, and $\mathcal{P}$ adds (a, v, c') to $\mathcal{W}^*$.

Check: $\mathcal{P}$ holds list $\mathcal{W}^*$ of entries (a, v, c), and $\mathcal{P}$ and $\mathcal{V}$ hold list $\mathcal{W}$ of entries $([a], [v], c)$ and list $\mathcal{R}$ of entries $([a], [v], [c])$.

1. For $(a, v, c) \in \mathcal{W}^*$: $\mathcal{P}$ and $\mathcal{V}$ compute $[a] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, a)$, $[v] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, v)$, and $[c] \leftarrow (\mathsf{Input}, c)$, adding $([a], [v], [c])$ to $\mathcal{R}$.
2. For $i \in [N]$, $\mathcal{P}$ and $\mathcal{V}$ compute $[x_i] \leftarrow \mathsf{Pack}(\mathcal{R}[i])$ and $[y_i] \leftarrow \mathsf{Pack}(\mathcal{W}[i])$.
3. $\mathcal{V}$ sends random field element $z \leftarrow \mathbb{F}$ to $\mathcal{P}$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[b] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Compute}, \mathsf{PolyCheck}, [x_1] - z, \ldots, [x_M] - z, [y_1] - z, \ldots, [y_N] - z)$.
5. $\mathcal{P}$ opens $[b]$ to $\mathcal{V}$, who aborts if $b \neq 1$.

Figure 5.4: Protocol for private read-only RAM access in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model.

We now show that the execution of Sim is statistically indistinguishable from the execution of $\mathcal{P}^*$ in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model. Since the view of $\mathcal{P}^*$ is perfectly simulated, we need only show that the output of $\mathcal{V}$ is statistically indistinguishable. If $\mathcal{P}^*$ behaves honestly in every operation so that $\mathcal{P}^*$ only reads values that was inserted during initialization, then the output of $\mathcal{V}$ is the same in both the ideal world and the $\mathcal{F}_{\mathsf{zk}}$-hybrid model. We now consider the case where $\mathcal{P}^*$ tries to cheat. Consider a Read operation in which $\mathcal{P}^*$ tries to cheat. There is a single way $\mathcal{P}^*$ can cheat. This is by reading a value that is not present in the memory. There are however two ways $\mathcal{P}^*$ can still win, despite of this. The first way is by simply committing to some value v' which isn't present in $\mathcal{W}^*$ or $\mathcal{W}$, at least not on the given address. This results in $\mathcal{R}$ and $\mathcal{W}$ not being equal, i.e. $\mathcal{R}$ and $\mathcal{W}$ are not permutations of each other and yet the permutation check passes. This happens with probability at most $(N+T)/|\mathbb{F}|$ following the same reasoning as presented by Franzese et al. [FKL+21].

The second way is by in addition to committing to a wrong value, $\mathcal{P}^*$ also commits to a wrong challenge. Now, as long the verifier sends over this same commitment in the future, $\mathcal{P}^*$ can make up the error and still pass the check. We formalize this error in Theorem 18. The probability of succes in this case, is $1/\mathbb{F}$.

Combining these, we get that Sim diverges from the ideal world with statistical probability $(N+T+1)/|\mathbb{F}|$, completing the proof. □

5.3.2 Read and Write Access to the Memory

We now present the full version of our protocol allowing for private reads and writes from the shared memory, without leaking neither the operation nor the address or the value.

Informally, we extend the protocol described in subsection 5.3.1 which only allowed for read operations, by adding back the arithmetic function f. Note that this function takes as input two values v, v' as well as an operation op which is either Read or Write (represented by 0 or 1, respectively). On Read, the function outputs v and on Write the function outputs v'.

A formal description of the functionality $\mathcal{F}_{\mathsf{zk\text{-}array}}$ is given in Figure 5.2 and a description of our protocol $\Pi_{\mathsf{zk\text{-}ram}}$ realizing the functionality $\mathcal{F}_{\mathsf{zk\text{-}array}}$ is given in Figure 5.5.

Theorem 18. *Protocol $\Pi_{\mathsf{zk\text{-}ram}}$ realizes the functionality $\mathcal{F}_{\mathsf{zk\text{-}array}}$ in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model with statistical error $(N+T+1)/|\mathbb{F}|$, where T is the number of read/write operations and N is the size of memory.*

Proof. We build a simulator Sim interacting with the $\mathcal{F}_{\mathsf{zk\text{-}array}}$ functionality as follows. Let $\mathcal{P}^*$ be the corrupted prover; Sim runs $\mathcal{P}^*$ as a subroutine and runs as follows:

- **Initialize**: For $i \in [N]$, Sim constructs $([a_i], [v_i], [c_i])$ from the calls $\mathcal{P}^*$ makes to $\mathcal{F}_{\mathsf{zk}}(\mathsf{Const})$. Sim then sends $(\mathsf{Initialize}, N, T)$ to $\mathcal{F}_{\mathsf{zk\text{-}array}}$.
- **Access**: On input $[\mathsf{op}]$, $[a]$, $[v']$, let v and c be the values $\mathcal{P}^*$ sends to $\mathcal{F}_{\mathsf{zk}}(\mathsf{Fix})$, and let y be the output of $\mathcal{F}_{\mathsf{zk}}(\mathsf{Compute})$. Sim sends a random challenge $c' \leftarrow \mathbb{F}$ to

Protocol $\Pi_{\mathsf{zk\text{-}ram}}$

Parameters:

- N: Size of memory.
- $\mathbb{F}$: Field to use in protocol.

Common circuits:

- f: Arithmetic circuit defined as follows:
  $f(v, v', \mathsf{op}) = v$ if $\mathsf{op} = \mathsf{Read}$ else v'.
- $\mathsf{PolyCheck}$: Arithmetic circuit defined as follows:
  $\mathsf{PolyCheck}(x_1, \ldots, x_M, y_1, \ldots, y_M) = 1$ if and only if $\prod x_i = \prod y_i$.

$\mathsf{Initialize}()$:

1. For $i \in [N]$: $\mathcal{P}$ and $\mathcal{V}$ compute $([a], [v], [c]) \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Const}, (i, 0, 0))$, storing the result in list $\mathcal{W}$.
2. $\mathcal{P}$ and $\mathcal{V}$ set list $\mathcal{R} = \emptyset$. $\mathcal{P}$ sets list $\mathcal{W}^* = \mathcal{W}$.

$\mathsf{ReadAndWriteBack}([a], [v'], [\mathsf{op}])$:

1. $\mathcal{P}$ finds entry $(a, v, c) \in \mathcal{W}^*$ and removes it.
2. $\mathcal{P}$ computes $[v] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, v)$ and $[c] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, c)$.
3. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [v], [c])$ to $\mathcal{R}$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[y] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Compute}, f, [v], [v'], [\mathsf{op}])$.
5. $\mathcal{V}$ sends random challenge $c' \leftarrow \mathbb{F}$ to $\mathcal{P}$.
6. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [y], c')$ to W, and $\mathcal{P}$ adds (a, y, c') to W^*.

Check: $\mathcal{P}$ holds list $\mathcal{W}^*$ of entries (a, v, c), and $\mathcal{P}$ and $\mathcal{V}$ hold list $\mathcal{W}$ of entries $([a], [v], c)$ and list R of entries $([a], [v], [c])$.

1. For $(a, v, c) \in \mathcal{W}^*$: $\mathcal{P}$ and $\mathcal{V}$ compute $[a] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, a)$, $[v] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, v)$, and $[c] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Input}, c)$, adding $([a], [v], [c])$ to R.
2. $\mathcal{V}$ sends random field element $z \leftarrow \mathbb{F}$ to $\mathcal{P}$.
3. For $i \in [N]$, $\mathcal{P}$ and $\mathcal{V}$ compute $[x_i] \leftarrow \mathsf{Pack}(\mathcal{R}[i])$ and $[y_i] \leftarrow \mathsf{Pack}(\mathcal{W}[i])$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[b] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Compute}, \mathsf{PolyCheck}, [x_1] - z, \ldots, [x_M] - z, [y_1] - z, \ldots, [y_N] - z)$.
5. $\mathcal{P}$ opens $[b]$ to $\mathcal{V}$, who aborts if $b \neq 1$.

Figure 5.5: Protocol for private read/write RAM access in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model.

$\mathcal{P}^*$, and adds (a, v, c) to R, (a, y, c') to W, and (a, y, c') to W^*. Finally, Sim sends $(\mathsf{Access}, [\mathsf{op}], [a], [v])$ to $\mathcal{F}_{\mathsf{zk\text{-}array}}$ and receives $[y]$.

- **Check**: For $i \in |\mathcal{W}^*|$, Sim obtains a_i, v_i, and c_i from $\mathcal{F}_{\mathsf{zk}}(\mathsf{Fix})$ and adds (a_i, v_i, c_i) to $\mathcal{R}$. Next, Sim sends a random challenge $z \leftarrow \mathbb{F}$ to $\mathcal{P}^*$.

  For $i \in [N]$, let $x_i \leftarrow \mathsf{Pack}(\mathcal{R}[i])$ and $y_i \leftarrow \mathsf{Pack}(\mathcal{W}[i])$. Sim obtains the values $x_i - z$ send by $\mathcal{P}^*$ to $\mathcal{F}_{\mathsf{zk}}(\mathsf{Compute})$. If $\mathsf{PolyCheck}(x_1 - z, \ldots, x_N - z) \neq 1$, Sim sends cheating to $\mathcal{F}_{\mathsf{zk\text{-}array}}$ and aborts, outputting whatever $\mathcal{P}^*$ outputs. Otherwise, Sim sends continue to $\mathcal{F}_{\mathsf{zk\text{-}array}}$ and halts, outputting whatever $\mathcal{P}^*$ outputs.

We now show that the execution of Sim is statistically indistinguishable from the execution of $\mathcal{P}^*$ in the $\mathcal{F}_{\mathsf{zk}}$-hybrid model. Since the view of $\mathcal{P}^*$ is perfectly simulated, we need only show that the output of $\mathcal{V}$ is statistically indistinguishable.

Consider a ReadAndWriteBack operation in which $\mathcal{P}^*$ tries to cheat. There are several possibilities:

- R and W are permutations of each other and yet memory consistency is violated (i.e., $\mathcal{P}^*$ successfully reads a value that is not equal to a previously written value). This can happen in two ways. Let $([a], [v], c)$ be a tuple previously stored in W, and consider the tuple $([a], [y], [c'])$ committed to by $\mathcal{P}^*$ and stored in $\mathcal{R}$.

  1. If $v \neq y$ but $c = c'$ then $\mathcal{P}^*$ gets caught in the permutation check.
  2. If $v \neq y$ and $c \neq c'$ then $\mathcal{P}^*$ successfully cheats if the verifier provided challenge, in a subsequent round, $c^\dagger$ is such that $c' = c^\dagger$ while the prover is trying to write y. This happens with probability $1/|\mathbb{F}|$.

- $\mathcal{R}$ and $\mathcal{W}$ are not permutations of each other and yet the permutation check passes (this happens if $\mathcal{P}^*$ commits to a wrong value or tries to use an address $a > N$). This happens with probability at most $(N + T)/|\mathbb{F}|$ following the same reasoning as presented by Franzese et al. [FKL+21].

Combining these, we get that Sim diverges from the ideal world with statistical probability $(N + T + 1)/|\mathbb{F}|$, completing the proof. □

Optimizations

Our first optimization is to utilize both the read and the write of ReadAndWriteBack instead of building Read and Write *on top* of ReadAndWriteBack. This reveals the operation, with the benefit of making the protocol twice as efficient, provided that the overall proof statement allows for this RAM interface.

Our second optimization is that ReadAndWriteBack need not be limited to implementing a Read or Write function, but rather can be called with *any* function, as long as the prover and verifier both agree on it. This allows for more abstract functions to be computed within a single memory access. While this has applications when the function f is private, we imagine that the biggest improvement comes when the overall

function being computed is public. Now, as the prover is not interested in hiding f, ReadAndWriteBack could take a function which could contain public *or* private values. For example, it would be possible for the function to be $f = x \mapsto x+1$ so that the overall call to ReadAndWriteBack would be $\mathsf{ReadAndWriteBack}([a], x \mapsto x+1)$ which represents a single operation to increment the value at a memory address. Likewise, it is also valid to compute, given $[x']$, $\mathsf{ReadAndWriteBack}([a], x \mapsto x+x')$.

Taking this further, the function passed to ReadAndWriteBack can be more than just a *pure* computation. It can also invoke ReadAndWriteBack itself[4]. As a result, there can be several operations in-flight at any given time, making room for concurrent memory accesses in an otherwise sequential computation. Furthermore, these generalizations may also allow the prover to reduce the number of memory operations performed.

5.3.3 Reducing Interaction of Our Protocol

We now present a modification of our protocol that removes the need for the verifier to send a challenge per-access to the prover. In this approach, the verifier provides an initial seed before any RAM access, and then the remaining protocol is locally computable by the prover using a hash function keyed by the seed. Note that the protocol can be made either fully non-interactive by using the Fiat-Shamir transformation [FS87] to generate the verifier's seed, *or* be used in a batch setting, where the verifier sends a seed only after B accesses, where B is the batch size.

We present this non-interactive protocol in Figure 5.6. The protocol makes use of a particular instantiation of $\mathcal{F}_{\mathsf{zk}}$ that modifies the Input operation as follows:

> On input (Input, x) from $\mathcal{P}$, store x and send $[x]$, $x^{\mathsf{corr}} = x - r$ *for a randomly generated* r*, and* $[r]$ to both parties. *We call* x^{corr} *the correlation value of* $[x]$*.*

We denote this modified functionality as $\mathcal{F}'_{\mathsf{zk}}$, and use the notation $([c], c^{\mathsf{corr}}) \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, c)$ when knowing the correlation value is important for the protocol[5] (otherwise we use $[c] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, c)$). This functionality can be instantiated by existing VOLE-based ZK protocols, such as Quicksilver [YSWW21a] and Mac'n'Cheese [BMRS21a].

Given $\mathcal{F}'_{\mathsf{zk}}$, the only change compared to Figure 5.5 is that now, instead of the prover receiving the challenge from the verifier during each RAM access, it computes the challenge as a hash of the verifier's seed sent as part of Initialize and the correction value for the prior challenge that the prover commits to in Step 2 of ReadAndWriteBack: that is, challenge $c'_{i+1} = H(\mathsf{seed}, c_i^{\mathsf{corr}})$. seed is set to c'_{i+1} for the generation of the next challenge.

Theorem 19. *Protocol* $\Pi_{\mathsf{NIzk\text{-}ram}}$ *in Figure 5.6 securely realizes the functionality* $\mathcal{F}_{\mathsf{zk\text{-}array}}$ *in the* $\mathcal{F}'_{\mathsf{zk}}$*-hybrid model with statistical error* $(N+T+1)/|\mathbb{F}| + 2^{-\rho} + (1-(1-2^{-\rho})^T)$*, assuming* H *is a random oracle with an output in* $\{0,1\}^\rho$*.*

---

[4] So long as this invocation targets an address that is distinct from the target address of any other in-flight ReadAndWriteBack operation.

[5] We do not utilize $[r]$ in the protocol; however, we do use it in the proof.

Protocol $\Pi_{\mathsf{NIzk\text{-}ram}}$

Parameters:

- N: Size of memory.
- $\mathbb{F}$: Field to use in protocol.
- H: Hash function modeled as a random oracle.
- ρ: Statistical security parameter.

Common circuits:

- f: Circuit defined as follows: $f(v, v', \mathsf{op}) = v$ if $\mathsf{op} = \mathsf{Read}$ else v'.
- $\mathsf{PolyCheck}$: Circuit defined as follows: $\mathsf{PolyCheck}(x_1, \ldots, x_M, y_1, \ldots, y_M) =$ 1 if and only if $\prod x_i = \prod y_i$.

$\mathsf{Initialize}()$:

1. For $i \in [N]$: $\mathcal{P}$ and $\mathcal{V}$ compute $([a], [v], [c]) \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Const}, (i, 0, 0))$, storing the result in list W.
2. $\mathcal{P}$ and $\mathcal{V}$ set list $R = \emptyset$. $\mathcal{P}$ sets list $W^* = W$.
3. $\mathcal{V}$ samples a random seed from $[2^\rho]$ and sends seed to $\mathcal{P}$.

$\mathsf{ReadAndWriteBack}([a], [v'], [\mathsf{op}])$:

1. $\mathcal{P}$ finds entry $(a, v, c) \in W^*$.
2. $\mathcal{P}$ computes $[v] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, v)$ and $([c], c^{\mathsf{corr}}) \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, c)$.
3. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [v], [c])$ to R.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[y] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Compute}, f, [v], [v'], [\mathsf{op}])$.
5. $\mathcal{P}$ and $\mathcal{V}$ compute a random challenge $c' \leftarrow H(\mathsf{seed}, c^{\mathsf{corr}})$.
6. $\mathsf{seed} \leftarrow c'$.
7. $\mathcal{P}$ and $\mathcal{V}$ add $([a], [y], c')$ to W, and $\mathcal{P}$ adds (a, y, c') to W^*.

Check: $\mathcal{P}$ holds list W^* of entries (a, v, c), and $\mathcal{P}$ and $\mathcal{V}$ hold list W of entries $([a], [v], c)$ and list R of entries $([a], [v], [c])$.

1. For $(a, v, c) \in W^*$: $\mathcal{P}$ and $\mathcal{V}$ compute $[a] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, a)$, $[v] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, v)$, and $[c] \leftarrow \mathcal{F}'_{\mathsf{zk}}(\mathsf{Input}, c)$, adding $([a], [v], [c])$ to R.
2. $\mathcal{V}$ sends random field element $z \leftarrow \mathbb{F}$ to $\mathcal{P}$.
3. For $i \in [N]$, $\mathcal{P}$ and $\mathcal{V}$ compute $x_i \leftarrow \mathsf{Pack}(R[i])$ and $y_i \leftarrow \mathsf{Pack}(W[i])$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[b] \leftarrow \mathcal{F}_{\mathsf{zk}}(\mathsf{Compute}, \mathsf{PolyCheck}, [x_1] - z, \ldots, [x_M] - z, [y_1] - z, \ldots, [y_N] - z)$.
5. $\mathcal{P}$ opens $[b]$ to $\mathcal{V}$, who aborts if $b \neq 1$.

Figure 5.6: Protocol for private read/write RAM access in the $\mathcal{F}'_{\mathsf{zk}}$-hybrid model which avoids the verifier challenge during each RAM access.

Proof. (Sketch) The proof is largely the same as that for Theorem 18, except now the challenges are not being sent by the verifier but rather generated based on the correction values sent by the *prover*.

The prover can cheat if it can fix some challenge c_i equal to some future challenge c_j. We show that the probability of this is $2^{-\rho}$. First, note that the correlation value c_i^{corr} binds c_i. This is because $c_i^{\mathsf{corr}} = c_i - r$ for some randomly generated r. That is, if a malicious prover tries to open $[c_i]$ to some value $c_i^* \neq c_i$, the verifier will detect this, since $[r] + c_i^{\mathsf{corr}} \neq c_i^*$.

Let $c_i = H(\mathsf{seed}, c_*^{\mathsf{corr}})$, where c_*^{corr} denotes the correlation value used to compute c_i. In order for the prover to find a c_j equal to c_i, it must compute $c_j = H(\mathsf{seed}, c_{j-1}^{\mathsf{corr}})$ for some valid correlation value c_{j-1}^{corr}.

Note that c_{j-1}^{corr} itself is associated with some challenge c_{j-1} computed as $H(\mathsf{seed}, c_{j-2}^{\mathsf{corr}})$, etc. Unrolling this, we find that the prover must compute

$$c_j = H(H(\dots H(\dots, c_i^{\mathsf{corr}}) \dots, c_{j-2}^{\mathsf{corr}}), c_{j-1}^{\mathsf{corr}}).$$

However, as H is a random oracle this only holds with probability $2^{-\rho}$.

An interesting aspect of the above, is that the analysis relies on the fact that the prover won't know when it could've cheated. If the collission happens immediatly so that $c_i = H(\mathsf{seed}, c_{i-1}^{\mathsf{corr}})$ for $c_i = c_{i-1}$, then the prover can take advantage of this in similar fashion to the above attack. Now, as already noted, the probability of finding a single collision is $2^{-\rho}$, thus the probability of there being no collisions in T memory operations is $(1 - 2^{-\rho})^T$. Using this, we find that the probability of the prover immediatly hitting a collision over T memory operations is $1 - (1 - 2^{-\rho})^T$.

Noting the above two attacks, the prover has an additional $2^{-\rho} + (1 - (1 - 2^{-\rho})^T)$ probability of cheating in the noninteractive protocol compared to the interactive variant. □

5.4 Asserting Permutations

Our protocol utilizes a permutation check originally due to Neff [Nef01] and used by Franzese et al. [FKL+21] in their RAM approach. We describe two variants of this check: one for individual field elements and one for tuples of field elements.

5.4.1 Asserting Permutations of Individual Values

Let $\mathcal{R} = ([r_1], \dots, [r_n])$ and $\mathcal{W} = ([w_1], \dots, [w_n])$ denote two lists in $\mathbb{F}^n$. Let $f(X) = \Pi_i(X - [r_i])$, $g(X) = \Pi_i(X - [w_i])$ be polynomials over $\mathbb{F}$. Let $z(X) = f(X) - g(X)$. Then $\mathcal{R}$ and $\mathcal{W}$ are permutations of each other if and only if $z(X)$ is the zero polynomial, since polynomials are identical under permutation of the roots [Gro09, Nef01]. Let $e \in \mathbb{F}$ be a randomly chosen field element. We can check that $z(e) = 0$ to check that $z(X)$ is the zero polynomial. Due to the Schwartz-Zippel lemma, the prover can cheat with probability $\frac{n}{|\mathbb{F}|}$. In our protocol, the verifier samples e and sends it to the prover, both

parties compute $z([e])$, and the parties open this commitment, allowing the verifier to check that it is zero.

Ensuring Soundness with Small Fields

We want $\mathbb{F}$ to be small in order to make the implementation of the zero knowledge proof scheme efficient. However, the smaller that we make $\mathbb{F}$, the smaller $\mathcal{R}$ and $\mathcal{W}$ must be in order to keep the permutation check sound. In particular, the permutation check is only sound if $n \leq 2^{-\rho} \cdot |\mathbb{F}_T|$, where ρ is the statistical security parameter.

We can increase the soundness of the permutation check at the cost of making it more computationally expensive.

Run the check twice If $n \leq 2^{\frac{-\rho}{2}} \cdot |\mathbb{F}|$, then running the above polynomial check twice, with two *different* random verifier challenges will be sound. This comes at the cost of double the multiplications of the simple polynomial check.

Karatsuba multiplication If $n \leq 2^{-\rho} \cdot |\mathbb{F}|^2$, then we want to run the polynomial check over the Galois extension field $\mathbb{F}^2$. To do this, we can treat commitments over $\mathbb{F}^2$ as pairs of commitments over $\mathbb{F}$. That is, $[x]_{\mathbb{F}^2} \mapsto ([x_L]_{\mathbb{F}}, [x_H]_{\mathbb{F}})$, where x_L and x_H are the high and low halves of x respectively. Pairwise linear operations on these commitment pairs are homomorphic: $c \cdot ([a]_{\mathbb{F}^2} + [b]_{\mathbb{F}^2}) = [c \cdot (a+b)]_{\mathbb{F}^2} \mapsto (c_L \cdot ([a_L]_{\mathbb{F}} + [b_L]_{\mathbb{F}}), c_H \cdot ([a_H]_{\mathbb{F}} + [b_H]_{\mathbb{F}})) = ([(c \cdot (a+b))_L]_{\mathbb{F}}, [(c \cdot (a+b))_H]_{\mathbb{F}})$.

Multiplications of commitments of $\mathbb{F}^2$ can be lowered into multiplication of commitments of $\mathbb{F}$ using Karatsuba multiplication: $([x]_{\mathbb{F}_T^2} \cdot [y]_{\mathbb{F}_T^2}) \mapsto ([z_1]_{\mathbb{F}_T}, [z_0 - z_2]_{\mathbb{F}_T})$ where $z_0 = x_L \cdot y_L$, $z_1 = (x_H + x_L)(y_H + y_L) - z_2 - z_0$, and $z_2 = x_H \cdot y_H$.

This approach triples the number of $\mathbb{F}$ multiplications required, but squares the size of the permutation allowed, while still being sound. This process can be repeated, as needed, as the number of memory operations grows[6].

5.4.2 Asserting Permutations of Tuples

For our protocol, *just* asserting that lists of individual field elements are permutations is not enough—we need to assert that lists of *tuples* of fields elements are permutations. To do this we pack our two lists of tuples into two lists of individual field elements, which we feed into the above single value permutation assertion. There are two ways we can implement this packing operation. In practice, we use both together. Below we describe these two approaches, using the field notation for VOLE-based correlations introduced in section 5.2; namely we use $\mathbb{F}_T$ and $\mathbb{F}_V$, where $\mathbb{F}_V$ is a subfield of $\mathbb{F}_T$.

[6] In practice, given a statistical security parameter of 2^{-40} (as a lower bound for $|\mathbb{F}|$), this would allow 2^{40} memory operations without needing to repeat this process, which should be enough for most practical applications.

Using extension fields If $\mathbb{F}_V = \mathbb{F}_{p^r}$ is a *proper* subfield of $\mathbb{F}_T = \mathbb{F}_{p^{rk}}$ (as is the case when working in binary), then there is a linear isomorphism $L : \mathbb{F}_{p^r}^k \to \mathbb{F}_{p^{rk}}$. Because commitments over $\mathbb{F}_V$ can also be viewed as commitments over $\mathbb{F}_T$, both parties can leverage the linear homomorphism of commitments to locally compute L over commitments. Thus, we define $\mathsf{Pack}([x_1], \ldots, [x_k]) = [L(x_1, \ldots, x_k)]$. Because L is an isomorphism, two lists of $\mathbb{F}_V^k$ are permutations if and only if the Pack-ed lists are permutations. Thus, we can use above permutation check in tandem with this Pack definition.

Using random projections This approach was introduced by de Saint Guilhem et al. [DOTV22, §3.2.2]. After the prover has committed to lists $\mathcal{W}$ and $\mathcal{R}$ of n values in $\mathbb{F}_T^k$, the verifier can send the prover a random vector: $e_1, \ldots, e_k \in \mathbb{F}_T$. Then we define $\mathsf{Pack} : \mathbb{F}_T^k \to \mathbb{F}_T$ as $\mathsf{Pack}(\mathbf{x}) = \mathbf{x} \cdot \mathbf{e}$. We can then check that the Pack-ed versions of $\mathcal{W}$ and $\mathcal{R}$ are permutations of each other to check that the original $\mathcal{W}$ and $\mathcal{R}$ are permutations of each other.

5.5 Implementation and Experiments

We have implemented our protocol in Rust using the Swanky Library [Gal19]. We evaluate the performance of our most general variant, $\Pi_{\mathsf{NIzk\text{-}ram}}$. Note that $\Pi_{\mathsf{NIzk\text{-}ram}}$ works similarly to $\Pi_{\mathsf{zk\text{-}ram}}$ in case the batch size of $\Pi_{\mathsf{NIzk\text{-}ram}}$ is 1 and that $\Pi_{\mathsf{zk\text{-}ro\text{-}ram}}$ is cheaper than $\Pi_{\mathsf{zk\text{-}ram}}$ due to not having a multiplexer in the form of the function f.

We ran $\Pi_{\mathsf{NIzk\text{-}ram}}$ using the binary field $\mathbb{F}_2$ in the extension field $\mathbb{F}_{2^{63}}$ as well as the prime field $\mathbb{F}_p$ for $p = 2^{61} - 1$. We instantiated the VOLE protocol using Wolverine [WYKW21a] and we use QuickSilver [YSWW21a] for the ZK proofs for polynomials.

We optimized our implementation for small-to-mediums sized servers running in a data center. Our benchmarks reflect this setting—all benchmarks are run between two m6.i8xlarge AWS servers, each having a 3rd Generation Intel Xeon Platinum 8375C with 32 vCPUs and 128GiB of RAM. One of the servers is located in North Virginia (us-east-1) and the other in Oregon (us-west-2). The servers run with roughly 70ms of latency and 4Gbps of bandwidth.

When considering the boolean case, our implementation is capable of having addresses as well as values of arbitrary size (at the cost of using a larger field for the commitments). For our experiments we vary the bit size of the values as indicated by the concrete tables. This matches previous work [FKL+21], for easier comparison. Lastly, we set our statistical security parameter to $\rho = 40$.

Table 5.1 presents the performance of our approach when using $\mathbb{F}_2$ with extension field $\mathbb{F}_{2^{63}}$ and varying the memory size from 65536 to 1048576. We can see that the cost per-access is roughly 6.3–7.3 μs regardless of the memory size, and this includes the cost of VOLE.

Table 5.2 presents the performance of our approach when using $\mathbb{F}_{p^{61}-1}$ while varying the number of operations performed. This indicates a correlation between the number of calls compared to the memory size, underlining our theoretical contribution mentioned

Memory Size	2^{15}	2^{18}	2^{20}
Total per access (in μs)	6.72	6.3	7.3

Table 5.1: Timing $\Pi_{\mathsf{NIzk\text{-}ram}}$ when varying memory size with 2^{22} calls to ReadAndWriteBack over the field $\mathbb{F}_2$ with extension field $\mathbb{F}_{2^{63}}$. We represent the values by 16 bits. All times include both the cost of VOLE, the call to ReadAndWriteBack as well as the final consistency check.

Operations	2^{20}	2^{22}	2^{23}	2^{25}
Total per access (in μs)	7.87	4.11	3.48	3.22

Table 5.2: Timing $\Pi_{\mathsf{NIzk\text{-}ram}}$ when varying number of calls to ReadAndWriteBack with a memory size of 2^{16} over the field $\mathbb{F}_p$ for $p = 2^{61} - 1$. All times include both the cost of VOLE, the call to ReadAndWriteBack as well as the final consistency check.

in subsection 5.1.3, as we see the time drop from 7.87 μs to 3.22 μs, as we increase the number of calls to ReadAndWriteBack.

Table 5.3 presents the performance of our approach when using a binary word sizes of $16, 32$ and 64 bits (that is, each RAM entry is comprised of varying $\mathbb{F}_2$ elements). Here we fix the memory size to be 65536 and see a per-access cost of at most 5.32 μs regardless of bit size. We have run the protocol due to Franzese et al. [FKL+21] resulting in a per-access cost of 25.2 μs, when considering 32 bit values (Table 5.4), making our approach roughly 5× more efficient. We note that this discrepancy between our numbers and theirs likely stem from their numbers being reported when limiting the bandwidth to 100 Mbps (whereas we run our experiments with a bandwidth of 4 Gbps), alongside the standard caveat that we are comparing two distinct implementations.

Furthermore, it's worth noting that we utilize a different permutation check for all of our benchmarks, which does not utilize QuickSilver [YSWW21a]. This permutation check sees slightly better computation times, at the cost of slightly increasing communication.

5.5.1 Potential Side-Channel Timing Attack

We note that both our implementation and the prior work of Franzese et al. [FKL+21] can potentially be attacked through a timing side channel, allowing the verifier to learn

Bit size	16	32	64
Total per access (in μs)	4.06	4.50	5.32

Table 5.3: Timing $\Pi_{\mathsf{NIzk\text{-}ram}}$ when varying the bit size of the values with 2^{23} calls to ReadAndWriteBack, a memory of size 2^{16} and using the extension field $\mathbb{F}_{2^{63}}$. All times include both the cost of VOLE, the call to ReadAndWriteBack as well as the final consistency check.

Bit size	16	32	64
Total per access (in μs)	22.5	25.2	32.0

Table 5.4: Timing the protocol by [FKL+21] when varying the bit size of values with 2^{23} memory operations, a memory of size 2^{16}. The times are for each memory operation.

information about the addresses being queried to RAM[7]. Considering our implementation first, in Step 1 of ReadAndWriteBack the prover needs to find an entry in $\mathcal{W}^*$ associated with a given address. We implement this lookup by indexing into an array with the address. However, the time it takes for the prover to read this value from the array may vary with the *value* of the address. This timing may leak information about the address to the verifier. This affects both $\Pi_{\mathsf{zk\text{-}ram}}$ and $\Pi_{\mathsf{NIzk\text{-}ram}}$[8].

Looking at Franzese et al. [FKL+21], their protocol and implementation suffers from a similar issue: when doing a RAM access the prover needs to look up the value stored at the provided address and commit to it. As in our implementation, the time required for this lookup may leak something about the address being queried.

One approach to addressing this issue is to utilize oblivious RAM (ORAM) on $\mathcal{W}^*$. This use of ORAM would be run entirely on the prover side and used as a means to hide the timing information from the verifier. Interestingly, this is not the standard use-case of ORAM, which is often viewed as a client-server protocol. An interesting future direction is to develop an ORAM protocol optimized for this non-standard setting.

We anticipate that, given the pipelining and concurrency optimizations we already have in place, switching to use ORAM (in the implementation of the prover) would have only minimal impact on performance.

[7] We have not experimentally validated that this timing side channel is indeed a concern in practice.

[8] If $\Pi_{\mathsf{NIzk\text{-}ram}}$ was made fully non-interactive by using Fiat-Shamir to sample the verifier's random seed alongside using a non-interactive instantiation of $\mathcal{F}'_{\mathsf{zk}}$ this attack would of course no longer be relevant.

Bibliography

[ABG+21] Diego F. Aranha, Carsten Baum, Kristian Gjøsteen, Tjerand Silde, and Thor Tunge. Lattice-based proof of shuffle and applications to electronic voting. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 227–251. Springer, Heidelberg, May 2021.

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

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011.

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

[Ale03] Michael Alekhnovich. More on average case vs approximation complexity. In *44th FOCS*, pages 298–307. IEEE Computer Society Press, October 2003.

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

[Bab85] László Babai. Trading group theory for randomness. In *17th ACM STOC*, pages 421–429. ACM Press, May 1985.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019.

[BBMH+21a] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and $\mathbb{Z}_{2^k}$. Cryptology ePrint Archive, Report 2021/750, 2021. https://eprint.iacr.org/2021/750.

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

[BBMHS22a] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Heidelberg, August 2022.

[BBMHS22b] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. Cryptology ePrint Archive, Report 2022/819, 2022. https://eprint.iacr.org/2022/819.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, January 2012.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[BCG+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 595–626. Springer, Heidelberg, December 2018.

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

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

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

[BCTV13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive arguments for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013. https://eprint.iacr.org/2013/879.

[BDG+13] Nir Bitansky, Dana Dachman-Soled, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, Adriana López-Alt, and Daniel Wichs. Why "Fiat-Shamir for proofs" lacks a proof. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 182–201. Springer, Heidelberg, March 2013.

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

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

[Ben64] V. E. Bene. Permutation groups, complexes, and rearrangeable connecting networks. *The Bell System Technical Journal*, 43(4):1619–1640, 1964.

[BFKL94] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 278–291. Springer, Heidelberg, August 1994.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state (extended version). Cryptology ePrint Archive, Report 2013/356, 2013. https://eprint.iacr.org/2013/356.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

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

[BMRS21a] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg.

[BMRS21b] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. 41st Annual International Cryptology Conference (CRYPTO 2021), 2021.

[BN19] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. Cryptology ePrint Archive, Report 2019/532, 2019. https://eprint.iacr.org/2019/532.

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

[Bou00] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444. Springer, Heidelberg, May 2000.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252. Springer, Heidelberg, December 2008.

[CD98] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 424–441. Springer, Heidelberg, August 1998.

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

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

[CDN15] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*, volume 1. Cambridge University Press, 2015.

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

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. Cryptology ePrint Archive, Report 1998/011, 1998. https://eprint.iacr.org/1998/011.

[Cha90] David Chaum. Showing credentials without identification transferring signatures between unconditionally unlinkable pseudonyms. In Jennifer Seberry and Josef Pieprzyk, editors, *AUSCRYPT'90*, volume 453 of *LNCS*, pages 246–264. Springer, Heidelberg, January 1990.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 302–321. Springer, Heidelberg, May 2005.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[CHP+23] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-knowledge proofs of real-world vulnerabilities, 2023.

[CKLR21] Geoffroy Couteau, Michael Klooß, Huang Lin, and Michael Reichle. Efficient range proofs with transparent setup from bounded integer commitments. Cryptology ePrint Archive, Report 2021/540, 2021. https://eprint.iacr.org/2021/540.

[CMM19] Núria Costa, Ramiro Martínez, and Paz Morillo. Lattice-based proof of a shuffle. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *FC 2019 Workshops*, volume 11599 of *LNCS*, pages 330–346. Springer, Heidelberg, February 2019.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.

[CPP17] Geoffroy Couteau, Thomas Peters, and David Pointcheval. Removing the strong RSA assumption from arguments over the integers. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 321–350. Springer, Heidelberg, April / May 2017.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 502–534, Virtual Event, August 2021. Springer, Heidelberg.

[DAT17] Thomas Debris-Alazard and Jean-Pierre Tillich. Statistical decoding. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 1798–1802. IEEE, 2017.

[DILO22] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 829–841. ACM Press, November 2022.

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

[DIO21a] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[DIO21b] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[DOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. Cryptology ePrint Archive, Report 2022/313, 2022. https://eprint.iacr.org/2022/313.

[DPSZ11] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

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

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors,

CRYPTO 2020, Part II, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

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

[EKM17] Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 486–514. Springer, Heidelberg, August 2017.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191. ACM Press, November 2021.

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

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

[Gal19] Galois, Inc. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky, 2019.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GHL21] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. Cryptology ePrint Archive, Report 2021/1397, 2021. https://eprint.iacr.org/2021/1397.

[GHL22] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 458–487. Springer, Heidelberg, May / June 2022.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th ACM STOC*, pages 365–377. ACM Press, May 1982.

[GMNO18] Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. Lattice-based zk-SNARKs from square span programs. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 556–573. ACM Press, October 2018.

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

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GNS21] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. Rinocchio: SNARKs for ring arithmetic. Cryptology ePrint Archive, Report 2021/322, 2021. https://eprint.iacr.org/2021/322.

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume 1. Cambridge University Press, 2001.

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 467–482. Springer, Heidelberg, June 2005.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 192–208. Springer, Heidelberg, August 2009.

[Gro11] Jens Groth. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 431–448. Springer, Heidelberg, December 2011.

[GT03] Shafi Goldwasser and Yael Tauman. On the (in)security of the Fiat-Shamir paradigm. Cryptology ePrint Archive, Report 2003/034, 2003. https://eprint.iacr.org/2003/034.

[Hac07] Peter Hackman. *Elementary Number Theory*. HHH Productions, Linköping, 2007.

[HK97] Shai Halevi and Hugo Krawczyk. MMH: Software message authentication in the Gbit/second rates. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 172–189. Springer, Heidelberg, January 1997.

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2055–2074. ACM Press, November 2020.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. ISC. Springer, Heidelberg, 2010.

[HMR15] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 150–169. Springer, Heidelberg, August 2015.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556. IEEE Computer Society Press, May 2021.

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

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

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

[IR90] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 8–26, New York, NY, 1990. Springer New York.

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

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*, pages 2–10. IEEE Computer Society Press, October 1990.

[Lin16] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://eprint.iacr.org/2016/046.

[Lip03] Helger Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. Cryptology ePrint Archive, Report 2003/105, 2003. https://eprint.iacr.org/2003/105.

[LLM+16] Benoît Libert, San Ling, Fabrice Mouhartem, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for matrix-vector relations and lattice-based group encryption. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 101–131. Springer, Heidelberg, December 2016.

[LNS20] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Shorter lattice-based zero-knowledge proofs via one-time commitments. Cryptology ePrint Archive, Report 2020/1448, 2020. https://eprint.iacr.org/2020/1448.

[LNS21] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Shorter lattice-based zero-knowledge proofs via one-time commitments. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 215–241. Springer, Heidelberg, May 2021.

[LWYY22] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. The hardness of LPN over any integer ring and field for PCG applications. Cryptology ePrint Archive, Report 2022/712, 2022. https://eprint.iacr.org/2022/712.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985. ACM Press, November 2021.

[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In Chandra Chekuri, Klaus Jansen, José D. P. Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 378–389, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Lyu08] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In Ronald Cramer, editor, *PKC 2008*, volume 4939 of *LNCS*, pages 162–179. Springer, Heidelberg, March 2008.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[MHOY21] Alexander Munch-Hansen, Claudio Orlandi, and Sophia Yakoubov. Stronger notions and a more efficient construction of threshold ring signatures. In Patrick Longa and Carla Ràfols, editors, *LATINCRYPT 2021*, volume 12912 of *LNCS*, pages 363–381. Springer, Heidelberg, October 2021.

[MN90] François Morain and Jean-Louis Nicolas. On cornacchias algorithm for solving the diophantine equation. 1990.

[MRS17] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 501–531. Springer, Heidelberg, April / May 2017.

[MRVW21] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 249–270. Springer, Heidelberg, March 2021.

[MSM+22] Chiara Marcolla, Victor Sucasas, Marc Manzano, Riccardo Bassoli, Frank H.P. Fitzek, and Najwa Aaraj. Survey on fully homomorphic encryption, theory, and applications. Cryptology ePrint Archive, Report 2022/1602, 2022. https://eprint.iacr.org/2022/1602.

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 116–125. ACM Press, November 2001.

[NNOB11] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Report 2011/091, 2011. https://eprint.iacr.org/2011/091.

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

[Pra62] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.

[RS86] Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39(S1):S239–S256, 1986.

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

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701717, oct 1980.

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

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

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

[Sha90] Adi Shamir. IP=PSPACE. In *31st FOCS*, pages 11–15. IEEE Computer Society Press, October 1990.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2022.

[WSH+14] Riad S. Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. Cryptology ePrint Archive, Report 2014/674, 2014. https://eprint.iacr.org/2014/674.

[WTs+18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.

[WYKW21a] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WYKW21b] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. 42nd IEEE Symposium on Security and Privacy (Oakland 2021), 2021.

[WYX+21a] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[WYX+21b] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 501–518, 2021.

[YSWW21a] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[YSWW21b] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials

over any field. 28th ACM Conference on Computer and Communications Security (CCS 2021), 2021.

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

[Zic17] Lior Zichron. Locally computable arithmetic pseudorandom generators. Masters thesis, School of Electrical Engineering, Tel Aviv University, 2017, 2017. http://www.eng.tau.ac.il/~bennyap/pubs/Zichron.pdf.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pages 216–226, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

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