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

Carsten Baum[0000-0001-7905-0198], Lennart Braun[0000-0001-9164-305X], Alexander Munch-Hansen[0000-0002-1482-0064], and Peter Scholl[0000-0002-7937-8422]

Aarhus University
{cbaum,braun,almun,peter.scholl}@cs.au.dk

Abstract. Zero-knowledge proof systems are usually designed to support computations for circuits over $\mathbb{F}_2$ or $\mathbb{F}_p$ for large p, but not for computations over $\mathbb{Z}_{2^k}$, which all modern CPUs operate on. Although $\mathbb{Z}_{2^k}$-arithmetic can be emulated using prime moduli, this comes with an unavoidable overhead. Recently, Baum et al. (CCS 2021) suggested a candidate construction for a designated-verifier zero-knowledge proof system that natively runs over $\mathbb{Z}_{2^k}$. Unfortunately, their construction requires preprocessed random vector oblivious linear evaluation (VOLE) to be instantiated over $\mathbb{Z}_{2^k}$. Currently, it is not known how to efficiently generate such random VOLE in large quantities.

In this work, we present 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. Our construction borrows ideas from recent protocols over finite fields, which we non-trivially adapt to work over $\mathbb{Z}_{2^k}$. Moreover, we show that the approach taken by the QuickSilver zero-knowledge proof system (Yang et al. 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 the previous zero-knowledge protocols suggested by Baum et al. Furthermore, we implement both our VOLE extension and our zero-knowledge proof system, and show that they can generate 13–50 million VOLEs per second for 64 bit to 256 bit rings, and evaluate 1.3 million 64 bit multiplications per second in zero-knowledge.

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 [25,26,7,16] 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. [22,8]) and MPC-in-the-head techniques (e.g. [2]), 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. [11], 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 [11] has only semi-honest security (or malicious security using expensive, generic 2-PC techniques). Later, dedicated maliciously secure protocols over fields were developed [12,25], 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 [16] 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 [7], 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 [5] 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 [23] 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 [17]. 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.

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 [12,25]. 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 [26] 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 [15] 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.

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 $M[x] = \Delta \cdot x + K[x]$ 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 $M[\overline{x}]$ on an input $\overline{x} \neq x$, for the same key $K[x]$, then the prover can recover the MAC key $\varDelta$ from the relation:

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

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 $M[\overline{x}]$ with probability $1/2$, since $M[x] - M[\overline{x}] \bmod 2^k$ now only depends on the least significant bit of $\varDelta$.

The SPD$\mathbb{Z}_{2^k}$ protocol [15] 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 $\varDelta$ 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 [5]. 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. [11], 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 [25], 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 [24,12]. When adapting this protocol to malicious security, our main technical challenge is that previous works over fields [12,25] 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 [25]: 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 [12], 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 [12,25], in this case, ours is more serious since entire noise coordinates can be leaked with probability 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 [25], we focus on using the "primal" form of LPN, which was also used for semi-honest VOLE over $\mathbb{Z}_{2^k}$ in [24]. While the "dual" form of LPN, as considered in [11,12,14], achieves lower communication costs (and does not rely on bootstrapping), it involves a more costly matrix multiplication, which is expensive to implement. In [12], dual-LPN was instantiated using quasi-cyclic codes to achieve $\widetilde{O}(n)$ 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 [14] 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 [26], 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$\mathbb{Z}_{2^k}$ [15] 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 Preliminaries

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 Vector OLE

Vector OLE (VOLE) is a two party functionality between a sender $\mathsf{P_S}$ and a receiver $\mathsf{P_R}$ to obtain correlated random vectors of the following form: $\mathsf{P_S}$ obtains two vectors $\mathbf{u}, \mathbf{w}$, and $\mathsf{P_R}$ 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 1.

As in SPD$\mathbb{Z}_{2^k}$ [15], we can implement $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ using the oblivious transfer protocol (OT) of [23]. 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 4 to repeatedly generate large batches to VOLEs.

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

All further (Init) queries are ignored.

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

1. If $\mathsf{P_R}$ 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 $\mathsf{P_S}$ is honest, sample $\mathbf{u} \in_R \mathbb{Z}^n_{2^\ell}$ and compute $\mathbf{w} := \varDelta \cdot \mathbf{u} + \mathbf{v} \in \mathbb{Z}_{2^\ell}$. 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 $\mathbf{v} := \mathbf{w} - \varDelta \cdot \mathbf{u} \in \mathbb{Z}^n_{2^\ell}$
3. Send $(\mathbf{u}, \mathbf{w})$ to $\mathsf{P_S}$ and $\mathbf{v}$ to $\mathsf{P_R}$.

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

Fig. 1. Ideal functionality VOLE over $\mathbb{Z}_{2^k}$.

2.3 Equality Test

In our work, we use an equality test functionality $\mathcal{F}_{\mathsf{EQ}}$ (Figure 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. [25]. 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.

2.4 Zero-Knowledge Proofs of Knowledge

In Figure 3 we provide an ideal functionality for zero-knowledge proofs. The functionality implies the standard definition of a ZKPoK as it is complete, knowledge sound and zero-knowledge.

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 $\mathcal{P}$.

Fig. 2. Ideal functionality for equality tests.

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

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

Fig. 3. Ideal functionality for zero-knowledge proofs for circuit satisfiability.

2.5 The LPN Assumption Over Rings

The *Learning Parity with Noise* (LPN) assumption [9] 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 [11,24].

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

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

There exists two flavours of the LPN assumption; the primal (Definition 1) and the dual (see e.g. [12]).

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 [1], 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 [4].

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 [14] 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.

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 [21], 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.[1] While [21] 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.[2] 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 [11,24,25], which according to [3,28] suffices to ensure that $\mathbf{A}$ has a large dual distance, which implies the LPN samples are unbiased [14].

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.

[1] This countermeasure was missing from the original version of this paper, before [21] was available.

[2] 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.

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 [27,24]. 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 [27,25].

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) 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 $\mathcal{F}_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$ in Figure 4. In the functionality, $\mathsf{P_S}$ obtains $\mathbf{u}, \mathbf{w} \in \mathbb{Z}_{2^\ell}^n \times \mathbb{Z}_{2^\ell}^n$, and $\mathsf{P_R}$ gets $\Delta, \mathbf{v} \in \mathbb{Z}_{2^s} \times \mathbb{Z}_{2^\ell}^n$. As in the full VOLE functionality $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ we allow $\mathsf{P_S}$ to attempt to guess Δ. Additionally, $\mathcal{F}_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$ also allows $\mathsf{P_R}$ to obtain leakage on the non-zero index:

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

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

Protocol Overview. Our protocol $\Pi_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$ (Figure 5) achieves active security using consistency checks inspired by the constructions from [12] and [25]. We now give a high-level overview.

As a setup, we assume functionalities $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$, $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{EQ}}$. For $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ we assume that $\mathsf{P_R}$ called (Init) already, thus setting Δ. Additionally, we require two pseudorandom generators (PRGs; with certain extra properties that we clarify

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 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 $\mathsf{P_S}$ 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 $\mathsf{P_R}$ 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 $(\mathsf{success}, \beta)$ to $\mathsf{P_R}$. If $\alpha \in I$ and $|I| > 1$, then send success to $\mathsf{P_R}$ 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 $\mathsf{P_S}$ and $\mathbf{v}$ to $\mathsf{P_R}$.

Global-key Query If $\mathsf{P_S}$ 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 $\mathsf{P_S}$. Otherwise, send abort to both parties and abort.

Fig. 4. Ideal functionality for a leaky single-point VOLE.

in Section 3.1) to create a GGM tree. Recall, the GGM construction [18] 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 $\mathsf{P_S}$ 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. $\mathsf{P_S}$ and $\mathsf{P_R}$ use a single VOLE from $\mathcal{F}^{\ell,s}_{\textsf{vole2k}}$ 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 $\mathbf{u}$, $\mathsf{P_R}$ 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 $\mathcal{F}_{\mathsf{OT}}$, $\mathsf{P_S}$ 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. $\mathsf{P_S}$ 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 $\mathsf{P_S}$ to learn $\mathbf{w}_\alpha = \Delta \cdot \mathbf{u}_\alpha + \mathbf{v}_\alpha$ without revealing α and β to $\mathsf{P_R}$. Using the output of the VOLE instance, if $\mathsf{P_R}$ computes $d \leftarrow \gamma - \sum_{j=1}^{n} \mathbf{v}_j$ and sends d to $\mathsf{P_S}$, then $\mathsf{P_S}$ 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 $\mathsf{P_S}$ (assuming that the hybrid functionalities are actively secure), a corrupted $\mathsf{P_R}$ 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, $\mathsf{P_R}$ has to send a random linear combination of these, over a binary extension field, with $\mathsf{P_S}$ 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 $\mathsf{P_S}$ 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 $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ 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 5. Before proving security of it, we first recap the Puncturable PRF from GGM construction and its security properties.

3.1 Checking Consistency of the GGM Construction

We use the GGM [19] 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

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

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

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

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

[a] Instead of sending the whole vector $\boldsymbol{\xi}$, $\textsf{P}_\textsf{S}$ can send a κ bit random seed which is then expanded with a PRG to obtain $\boldsymbol{\xi}$.
[b] Again, $\textsf{P}_\textsf{S}$ can send a short seed instead of $\boldsymbol{\chi}$.

Fig. 5. Protocol instantiating $\mathcal{F}^{\ell,s}_{\textsf{sp-vole2k}}$ in the $(\mathcal{F}^{\ell,s}_{\textsf{vole2k}}, \mathcal{F}_{\textsf{OT}}, \mathcal{F}_{\textsf{EQ}})$-hybrid model.

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 2 (Adapted from [12]). *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 $k\{i\} \in \mathcal{K}_p$.*
$\mathsf{PuncEval}$ *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 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[3] as defined below.

Definition 3. *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 $\varXi$. This algorithm, on input $n, \alpha, k\{\alpha\}$, a challenge ξ and a checking value $\varGamma$ outputs $\top$ or $\bot$.

Definition 4 (PPRF consistency). *Let F be a PPRF and let $\varXi$ 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. $\xi \in_R \varXi$
3. $\varGamma \leftarrow \mathcal{A}(1^\kappa, \mathsf{state}, \xi)$
4. *For all $\alpha \in [n]$, let* $\mathbf{v}^\alpha \leftarrow \mathsf{PuncEval}(1^\kappa, \alpha, k\{\alpha\})$.

[3] As noted in [12], this can be replaced with a weaker notion of right-half collision resistance, which is easier to achieve in practice.

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 $K_0^0 \leftarrow k$.
   (b) For each level $i \in [h]$, and for $j \in \{0, \ldots, 2^{i-1}-1\}$ compute $(K_{2j}^i, K_{2j+1}^i) \leftarrow \mathsf{G}(K_j^{i-1})$.
   (c) For $i \in [h]$, set $\overline{K}_0^i \leftarrow \bigoplus_{j=0}^{2^{i-1}-1} K_{2j}^i$ and $\overline{K}_1^i \leftarrow \bigoplus_{j=0}^{2^{i-1}-1} K_{2j+1}^i$.
   (d) For $j \in [2^h]$ compute $v_j, t_j \leftarrow \mathsf{G}'(K_{j-1}^h)$, and set $\mathbf{v} := (v_1, \ldots, v_{2^h})$ and $\mathbf{t} := (t_1, \ldots, t_{2^h})$.
   (e) Compute $\overline{K}_1^{h+1} \leftarrow \sum_{j \in [2^h]} t_i$.
   (f) Output $(\mathbf{v}, \mathbf{t}, (\overline{K}_0^i, \overline{K}_1^i)_{i \in [h]}, \overline{K}_1^{h+1})$.
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 $K_{\overline{\alpha}_1}^1 \leftarrow \overline{K}^1$.
   (b) For each level $i \in \{2, \ldots, h\}$:
      i. Let $x := \sum_{j=1}^{i-1} 2^{j-1} \cdot \alpha_{i-j}$
      ii. For $j \in \{0, \ldots, 2^{i-1}-1\} \setminus \{x\}$, compute $(K_{2j}^i, K_{2j+1}^i) \leftarrow \mathsf{G}(K_j^{i-1})$.
      iii. Compute $K_{2x+\overline{\alpha}_i}^i \leftarrow \overline{K}^i \oplus \bigoplus_{\substack{0 \le j < 2^{i-1} \\ j \neq x}} K_{2j+\overline{\alpha}_i}^i$.
   (c) For the last level $h+1$:
      i. For $j \in [2^h] \setminus \{\alpha\}$ compute $(v_j, t_j) \leftarrow \mathsf{G}'(K_{j-1}^h)$
   (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 $\perp$.

In comparison to Definition 2 $\mathsf{GGM.Gen}$ computes a compressed version of all keys. The pseudorandomness for GGM, as defined in Definition 2, follows from the standard pseudorandomness argument of the GGM construction [20,10,13].

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. We give the proof in the full version [6].

Theorem 5 (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 4 with probability at most* $2^{-(\sigma+1)}$.

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

Theorem 6. *The protocol $\Pi^{\ell,s}_{\textsf{sp-vole2k}}$ (Figure 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* $2^{-(\sigma+1)} + \mathsf{negl}(\kappa)$.

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 $\textsf{GGM}$ 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 $\textsf{GGM}$ 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 $\textsf{GGM}$ 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 $\textsf{GGM}$ 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.

The full proof of Theorem 6, together with a summary of the protocol complexity, can be found in the full version [6].

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.

4.1 Leaky Regular LPN Assumption

The assumption, below, translates the leakage from the single-point VOLE functionality (Figure 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 [12,25], 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 7. *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 $\mathbf{e} = (\mathbf{e}_1, \dots, \mathbf{e}_t) \leftarrow \mathbb{Z}_{2^\ell}^n$, 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.

4.2 Vector OLE Protocol

Our complete VOLE protocol is given in Figure 6. It realises the functionality $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$ (Figure 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 [25], 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 $\mathcal{F}_{\mathsf{sp\text{-}vole2k}}^{\ell,s}$ to generate $\mathbf{c} = (c_1, \dots, c_t)$ and $\mathbf{e} = (e_1, \dots, e_t)$ for the sender and $\mathbf{b} = (b_1, \dots, b_t)$ for the receiver. The sender then simply computes $\mathbf{x} \leftarrow \mathbf{u} \cdot \mathbf{A} + \mathbf{e} \in \mathbb{Z}_{2^r}^n$ and $\mathbf{z} \leftarrow \mathbf{w} \cdot \mathbf{A} + \mathbf{c} \in \mathbb{Z}_{2^r}^n$ and the receiver computes $\mathbf{y} = \mathbf{v} \cdot \mathbf{A} + \mathbf{b} \in \mathbb{Z}_{2^r}^n$. 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.[4]

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
- $\mathbf{A} \in \mathbb{Z}^{m \times n}_{2^\ell}$ is the generator matrix used in primal-LPN

Init This must be called by the parties first and is executed once.
1. $\mathsf{P_S}$ and $\mathsf{P_R}$ send (Init) to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$, and $\mathsf{P_R}$ receives $\Delta \in \mathbb{Z}_{2^s}$.
2. $\mathsf{P_S}$ and $\mathsf{P_R}$ send (Extend, m) to $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$. $\mathsf{P_S}$ receives $\mathbf{u}, \mathbf{w} \in \mathbb{Z}^m_{2^\ell}$, and $\mathsf{P_R}$ receives $\mathbf{v} \in \mathbb{Z}^m_{2^\ell}$, 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 $i \in [t]$, $\mathsf{P_S}$ and $\mathsf{P_R}$ 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 $\mathsf{P_S}$ and $\mathbf{b}_i$ to $\mathsf{P_R}$ such that $\mathbf{c}_i = \Delta \cdot \mathbf{e}_i + \mathbf{b}_i$ over $\mathbb{Z}^{n/t}_{2^\ell}$, and $\mathbf{e}_i \in \mathbb{Z}^{n/t}_{2^\ell}$ has exactly one entry invertible modulo 2^ℓ and zeros everywhere else.
2. Define $\mathbf{e} := (\mathbf{e}_1, \ldots, \mathbf{e}_t) \in \mathbb{Z}^n_{2^\ell}$, $\mathbf{c} := (\mathbf{c}_1, \ldots, \mathbf{c}_t) \in \mathbb{Z}^n_{2^\ell}$, and $\mathbf{b} := (\mathbf{b}_1, \ldots, \mathbf{b}_t) \in \mathbb{Z}^n_{2^\ell}$. Then $\mathsf{P_S}$ computes $\mathbf{x} := \mathbf{u} \cdot \mathbf{A} + \mathbf{e} \in \mathbb{Z}^n_{2^\ell}$, and $\mathbf{z} := \mathbf{w} \cdot \mathbf{A} + \mathbf{c} \in \mathbb{Z}^n_{2^\ell}$. $\mathsf{P_R}$ computes $\mathbf{y} := \mathbf{v} \cdot \mathbf{A} + \mathbf{b} \in \mathbb{Z}^n_{2^\ell}$.
3. $\mathsf{P_S}$ updates $\mathbf{u}, \mathbf{w}$ by setting $\mathbf{u} := \mathbf{x}[0 : m) \in \mathbb{Z}^m_{2^\ell}$ and $\mathbf{w} := \mathbf{z}[0 : m) \in \mathbb{Z}^m_{2^\ell}$, and outputs $(\mathbf{x}[m : n), \mathbf{z}[m : n)) \in \mathbb{Z}^\ell_{2^\ell} \times \mathbb{Z}^\ell_{2^\ell}$. $\mathsf{P_R}$ updates $\mathbf{v}$ by setting $\mathbf{v} := \mathbf{y}[0 : m) \in \mathbb{Z}^m_{2^\ell}$ and outputs $\mathbf{y}[m : n) \in \mathbb{Z}^\ell_{2^\ell}$.

Fig. 6. Protocol for VOLE over $\mathbb{Z}_{2^k}$ in the $\mathcal{F}^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$-hybrid model. Based on [25].

Theorem 8. *The protocol $\Pi^{\ell,s}_{\mathsf{vole2k}}$ in Fig. 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.*

The proof, given in the full version [6], is straightforward for the malicious sender, and for the malicious receiver we translate the protocol into an instance of primal LPN from Definition 1, which yields indistinguishability.

Communication Complexity When we instantiate the single-point VOLE with our protocol $\Pi^{\ell,s}_{\mathsf{sp\text{-}vole2k}}$ from Section 3, use the equality test sketched in Section 2.3, and Silent OT [12,27,14], our VOLE extension protocol $\Pi^{\ell,s}_{\mathsf{vole2k}}$ with

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

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 the full version [6].

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 [26], similar to LPZK [16], 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 $\mathbb{Z}_{2^k}$.

5.1 Bounding the Number of Solutions to Quadratic Equations

In the following theorem, we analyze a security game that corresponds to our amortized check for verifying t multiplications. At the core of this, we need to upper bound the number of solutions to quadratic equations in $\mathbb{Z}_{2^\ell}$, where both the coefficients and solutions are bounded in certain ways.

Theorem 9. *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 $\delta_0, \ldots, \delta_t \in \mathbb{Z}$ such that not all δ_i for $i > 0$ are $0 \bmod 2^k$.*
3. *$\mathcal{C}$ chooses $\chi_1, \ldots, \chi_t \leftarrow \mathbb{Z}_{2^s}$ 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 9 follows a similar way as Lemma 1 of [15]. 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. The full proof can be found in the full version [6], where we also show the following corollary.

Corollary 10. *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 9 with probability at most $2^{-\sigma}$.*

5.2 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 [5] 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 $K[x], M[x] \in_R \mathbb{Z}_{2^\ell}$ with $\ell \geq k + s$ so that

$$K[x] = M[x] + \widetilde{x} \cdot \Delta \pmod{2^\ell} \tag{1}$$

holds for $\widetilde{x} = x \pmod{2^k}$. Here the prover knows $\widetilde{x}$ and $M[x]$, and the verifier knows Δ and $K[x]$. To open the commitment, the prover reveals $\widetilde{x}, K[x]$ 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 $\widetilde{x}, K[x]$ and $M[x]$ 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 $[y]$.

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 $[r_1], \ldots, [r_{n+t+2}]$ 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 (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

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

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

QuarkSilver Π_{QS}^k

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 $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$, and $\mathcal{V}$ receives $\Delta \in \mathbb{Z}_{2^s}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Extend}, \mathrm{n} + \mathrm{t} + 2)$ to $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$, 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 $K[w_i] = M[w_i] + \widetilde{w}_i \cdot \Delta$ for $i \in \{\alpha, \beta, \gamma\}$.
   - $\mathcal{P}$ computes $A_{0,i} := M[w_\alpha] \cdot M[w_\beta] \in \mathbb{Z}_{2^\ell}$ and $A_{1,i} := \widetilde{w}_\alpha \cdot M[w_\beta] + \widetilde{w}_\beta \cdot M[w_\alpha] - M[w_\gamma] \in \mathbb{Z}_{2^\ell}$.
   - $\mathcal{V}$ computes $B_i := K[w_\alpha] \cdot K[w_\beta] - \Delta \cdot K[w_\gamma] \in \mathbb{Z}_{2^\ell}$.
4. $\mathcal{P}$ and $\mathcal{V}$ run the following check:
   (a) Set $A_0^* := M[o]$, $A_1^* := \widetilde{o}$, and $B^* := K[o]$ 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 $M[z]$ to $\mathcal{V}$ who checks that $\widetilde{z} = 1 \pmod{2^k}$ and $K[z] = M[z] + \widetilde{z} \cdot \Delta$. $\mathcal{V}$ outputs true iff the check passes, and false otherwise.

Fig. 7. Zero-knowledge protocol for circuit satisfiability in the $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$-hybrid model with $s := \sigma + \log(\sigma) + 3$ and $\ell := k + 2s$ for statistical security parameter σ.

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 $\chi \leftarrow \mathbb{Z}_{2^s}^t$, 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 7.

Security of the QuarkSilver Protocol

Theorem 11. *The protocol Π_{QS}^k (Figure 7) securely realizes the functionality $\mathcal{F}_{\mathsf{ZK}}^k$ in the $\mathcal{F}_{\mathsf{vole2k}}^{\ell,s}$-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 [26], the structure of our proof is also similar. The main difference, lies in the proof of soundness of the multiplication check. We will sketch the argument briefly, while the full proof of Theorem 11 can be found in the full version [6].

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 committed values in $[w_\alpha], [w_\beta], [w_\gamma]$ and $e_i \in \mathbb{Z}_{2^\ell}$ is a possible error. Suppose that not all $e_i = 0 \pmod{2^k}$ for $i \in [t]$. Then

$$K[w_\gamma] = M[w_\gamma] + \widetilde{w}_\gamma \cdot \Delta = M[w_\gamma] + (\widetilde{w}_\alpha \cdot \widetilde{w}_\beta) \cdot \Delta + e_i \cdot \Delta \pmod{2^\ell}$$

and (also modulo 2^ℓ)

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

where $A_{i,0}$ and $A_{i,1}$ are as above the values that an honest $\mathcal{P}$ would send. The equations for all gates are aggregated using a random linear combination:

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

Here, U, V denote the values that an honest $\mathcal{P}$ would send. The corrupted $\mathcal{P}^*$ may choose to send $U' := U + e_U$ and $V' := V + e_V$ instead, and $\mathcal{V}$ accepts if $W = U' + V' \cdot \Delta$ holds. Rearranging Equation 2, we get that $\mathcal{V}$ accepts if

$$0 = e_U + e_V \cdot \Delta + \left(\sum\nolimits_{i\in[t]} \chi_i \cdot e_i\right) \cdot \Delta^2 \pmod{2^\ell} \tag{3}$$

holds. The key observation is that the steps in the protocol correspond exactly to the game defined in Theorem 9 and the dishonest prover wins the game, i.e., cheats successfully, if Equation (3) holds. By Corollary 10 the probability that this happens is at most $2^{-\sigma}$.

General Degree-2 Checks. Yang et al. [26] 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 5.1, 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 give the full protocol and its security proof in the full version [6].

6 Experiments

In this section we report on the performance of our VOLE protocol $\Pi^{r,s}_{\mathsf{vole2k}}$ (Section 4) and our zero-knowledge proof system QuarkSilver (Section 5). We implemented the protocols in the Rust programming language using the *swanky* framework[5]. 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.

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.

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.

6.2 VOLE Experiments

In this section, we evaluate the performance of our VOLE protocol $\Pi^{\ell,s}_{\mathsf{vole2k}}$ (Section 4). We consider the setting of batch-wise VOLE extension: Given set of n_b

[5] swanky: `https://github.com/GaloisInc/swanky`

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 [24,27,25], 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 2.5, 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 [12,25] (see also full version [6]). Note that, as our variant of the regular LPN assumption (Definition 7) 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 $\Pi_{\textsf{sp-vole2k}}^{\ell,s}$ 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.

We chose LPN parameters targeting a level of $\kappa = 128$ bits of computational security, and used the approach of Boyle et al. [11] to estimate the hardness of the LPN problem. Recently, Liu et al. [21] 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. [21], 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 the full version [6].

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 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 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	$\mathsf{P_S} \to \mathsf{P_R}$	$\mathsf{P_R} \to \mathsf{P_S}$	total
	$m = 553\,600$, $t = 2\,186$, $n = 10\,558\,380$					
	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
40	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
	$m = 830\,800$, $t = 2\,013$, $n = 10\,835\,979$					
	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
80	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

Comparison with Wolverine. We compare the efficiency of our VOLE extension protocol with that of Wolverine [25]. 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, where we list our run-times in different bandwidth settings with the corresponding numbers given in [25]. 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.

Table 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 [25].

	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	—

Bandwidth Dependence. Table 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.

6.3 Zero-Knowledge Experiments

We explore at what rate our QuarkSilver protocol (Section 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 7) to commit to the results $w_{i,\gamma} := w_{i,\alpha} \cdot w_{i,\beta}$ of the multiplications.
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 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 3 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.

Table 3. 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	$\mathsf{P_S} \to \mathsf{P_R}$	$\mathsf{P_R} \to \mathsf{P_S}$	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

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. [26]. 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 $3\times$ 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 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. [5] – 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) under 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.

References

1. Alekhnovich, M.: More on average case vs approximation complexity. In: 44th FOCS. pp. 298–307. IEEE Computer Society Press (Oct 2003). `10.1109/SFCS.2003.1238204`

2. Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: Lightweight sublinear arguments without a trusted setup. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 2087–2104. ACM Press (Oct / Nov 2017). 10.1145/3133956.3134104
3. Applebaum, B., Damgård, I., Ishai, Y., Nielsen, M., Zichron, L.: Secure arithmetic computation with constant computational overhead. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 223–254. Springer, Heidelberg (Aug 2017). 10.1007/978-3-319-63688-7_8
4. Arora, S., Ge, R.: New algorithms for learning in presence of errors. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 403–415. Springer, Heidelberg (Jul 2011). 10.1007/978-3-642-22006-7_34
5. Baum, C., Braun, L., Munch-Hansen, A., Razet, B., Scholl, P.: Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 192–211. ACM Press (Nov 2021). 10.1145/3460120.3484812
6. Baum, C., Braun, L., Munch-Hansen, A., Scholl, P.: Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-ole and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. Cryptology ePrint Archive, Paper 2022/819 (2022), https://eprint.iacr.org/2022/819, Full Version
7. Baum, C., Malozemoff, A.J., Rosen, M.B., Scholl, P.: Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 92–122. Springer, Heidelberg, Virtual Event (Aug 2021). 10.1007/978-3-030-84259-8_4
8. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 701–732. Springer, Heidelberg (Aug 2019). 10.1007/978-3-030-26954-8_23
9. Blum, A., Furst, M.L., Kearns, M.J., Lipton, R.J.: Cryptographic primitives based on hard learning problems. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 278–291. Springer, Heidelberg (Aug 1994). 10.1007/3-540-48329-2_24
10. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (Dec 2013). 10.1007/978-3-642-42045-0_15
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press (Oct 2018). 10.1145/3243734.3243868
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press (Nov 2019). 10.1145/3319535.3354255
13. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar 2014). 10.1007/978-3-642-54631-0_29
14. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 502–534. Springer, Heidelberg, Virtual Event (Aug 2021). 10.1007/978-3-030-84252-9_17
15. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 769–798. Springer, Heidelberg (Aug 2018). 10.1007/978-3-319-96881-0_26

16. Dittmer, S., Ishai, Y., Ostrovsky, R.: Line-point zero knowledge and its applications. In: 2nd Conference on Information-Theoretic Cryptography (ITC 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
17. Ganesh, C., Nitulescu, A., Soria-Vazquez, E.: Rinocchio: SNARKs for ring arithmetic. Cryptology ePrint Archive, Report 2021/322 (2021), `https://eprint.iacr.org/2021/322`
18. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: 25th FOCS. pp. 464–479. IEEE Computer Society Press (Oct 1984). `10.1109/SFCS.1984.715949`
19. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM (JACM) **33**(4), 792–807 (1986)
20. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013). `10.1145/2508859.2516668`
21. Liu, H., Wang, X., Yang, K., Yu, Y.: The hardness of lpn over any integer ring and field for pcg applications. Cryptology ePrint Archive, Paper 2022/712 (2022), `https://eprint.iacr.org/2022/712`
22. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2111–2128. ACM Press (Nov 2019). `10.1145/3319535.3339817`
23. Scholl, P.: Extending oblivious transfer with low communication via key-homomorphic PRFs. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part I. LNCS, vol. 10769, pp. 554–583. Springer, Heidelberg (Mar 2018). `10.1007/978-3-319-76578-5_19`
24. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-OLE: Improved constructions and implementation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1055–1072. ACM Press (Nov 2019). `10.1145/3319535.3363228`
25. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 2021 IEEE Symposium on Security and Privacy. pp. 1074–1091. IEEE Computer Society Press (May 2021). `10.1109/SP40001.2021.00056`
26. Yang, K., Sarkar, P., Weng, C., Wang, X.: QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 2986–3001. ACM Press (Nov 2021). `10.1145/3460120.3484556`
27. Yang, K., Weng, C., Lan, X., Zhang, J., Wang, X.: Ferret: Fast extension for correlated OT with small communication. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1607–1626. ACM Press (Nov 2020). `10.1145/3372297.3417276`
28. Zichron, L.: Locally computable arithmetic pseudorandom generators. Master's thesis, School of Electrical Engineering, Tel Aviv University, 2017 (2017), `http://www.eng.tau.ac.il/~bennyap/pubs/Zichron.pdf`