Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits

Chenkai Weng
Northwestern University

Kang Yang*
State Key Laboratory of Cryptology

Jonathan Katz†
University of Maryland

Xiao Wang*
Northwestern University

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00056

***Abstract*—Efficient zero-knowledge (ZK) proofs for arbitrary boolean or arithmetic circuits have recently attracted much attention. Existing solutions suffer from either significant prover overhead (i.e., high memory usage) or relatively high communication complexity (at least κ bits per gate, for computational security parameter κ). In this paper, we propose a new protocol for constant-round interactive ZK proofs that simultaneously allows for an efficient prover with asymptotically optimal memory usage and significantly lower communication compared to protocols with similar memory efficiency. Specifically:**

- **The prover in our ZK protocol has linear running time and, perhaps more importantly, memory usage linear in the memory needed to evaluate the circuit non-cryptographically. This allows our proof system to scale easily to very large circuits.**
- **For statistical security parameter $\rho = 40$, our ZK protocol communicates roughly 9 bits/gate for boolean circuits and 2–4 field elements/gate for arithmetic circuits over large fields.**

Using 5 threads, 400 MB of memory, and a 200 Mbps network to evaluate a circuit with hundreds of billions of gates, our implementation ($\rho = 40, \kappa = 128$) runs at a rate of 0.45 μs/gate in the boolean case, and 1.6 μs/gate for an arithmetic circuit over a 61-bit field.

We also present an improved subfield Vector Oblivious Linear Evaluation (sVOLE) protocol with malicious security that is of independent interest.

I. Introduction

Zero-knowledge (ZK) proofs (of knowledge) [39], [37] are a fundamental cryptographic tool. They allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$, who holds a circuit $\mathcal{C}$, that the prover knows a witness w for which $\mathcal{C}(w) = 1$, without leaking any extra information. While ZK proofs for arbitrary circuits are possible [37], until recently such proofs were inefficient as they relied on reductions to generic NP-complete problems. Over the past decade, however, several ZK proof systems have been developed that yield far more efficient protocols. These include *zero-knowledge succinct non-interactive arguments of knowledge* (zk-SNARKs) [40], [32], [7], [9], [10], [20], [55], [8], [6], [54], ZK proofs based on Interactive Oracle Proofs (IOPs) and techniques from the setting of verifiable outsourcing [38], [58], [21], [63], ZK proofs following the "MPC-in-the-head" approach [46], [33], [23], [1], [48], [28], and a line of work constructing ZK proofs from garbled circuits (ZKGC) [47], [31], [61], [42]. Each of these works offers different tradeoffs between underlying assumptions (both computational hardness assumptions as well as setup assumptions), round complexity (in particular, whether the proof requires interaction or can be made non-interactive), expressiveness (e.g., whether the scheme natively handles boolean or arithmetic circuits), and efficiency. With regard to efficiency, measures of interest include the prover complexity (including time complexity and memory requirements), the verifier complexity, and the communication as a function of the circuit size.

One important factor is the memory overhead of ZK protocols. In particular, high memory requirements can impose a hard limit on the maximum circuit size that a protocol can support in practice. As shown in Table I, prior ZK proof systems can be characterized roughly as either (1) having short proofs (e.g., sublinear in the circuit size, or even sublinear in the length of a witness) but significant memory overhead for the prover as in the case of zk-SNARKs, IOP-based schemes, and some schemes following the MPC-in-the-head paradigm, or (2) imposing low memory overhead for the prover but having high communication complexity, as in the case of ZKGC schemes.

In this paper, we propose a new approach to ZK proofs that enables an extremely efficient prover in both running time and memory usage while having lower communication compared to the ZKGC approach that offers similar prover efficiency. As in the ZKGC approach, we obtain prover complexity—in terms of both time and memory usage—linear in the complexity required to evaluate the circuit non-cryptographically; this allows our ZK protocol to scale easily to very large circuits. At the same time, we achieve communication complexity that is more than an order of magnitude lower than what can be achieved using the ZKGC approach, while natively supporting boolean or arithmetic circuits. As compared to the other work in Table I, the main drawback of our protocol—shared by the ZKGC approach—is that it requires interaction. Our protocol does, however, offer a non-interactive *online* phase following an interactive offline phase that can be executed by the parties before the circuit is known.

A. Outline of Our Solution

Our ZK protocol (named Wolverine) can be separated into two phases: an interactive offline phase that can be executed by the prover and verifier before both the circuit and the witness are known, and an online phase that can be made non-interactive in the random-oracle model. We view the online phase as our main conceptual contribution, though we offer efficiency improvements for the offline phase as well.

†Work done as a consultant for Stealth Software Technologies, Inc.
*Corresponding authors

	Protocol Type	Spartan [54] zk-SNARK	Virgo [63] IOP-based	Ligero [1] MPC-in-the-head	[42] ZKGC	Wolverine sVOLE-based
Merkle tree (boolean circuit)	Prover time	55 s	53 s	400 s	7.3 s	11 s
	Verifier time	< 0.1 s	< 0.1 s	< 0.1 s	7.3 s	11 s
	Overall time	55 s	53 s	400 s	7.3 s	11 s
	Communication	$\leq$ 100 KB	253 KB	1.5 MB	182.2 MB	12.4 MB
	Prover memory	$\approx$ 7 GB	$\approx$ 1 GB	$\approx$ 5 GB	$\leq$ 400 MB	$\leq$ 400 MB
Matrix mult. (arithmetic circuit)	Prover time	677 s	64 s	–	–	320 s
	Verifier time	< 0.1 s	< 0.1 s	–	–	320 s
	Overall time	677 s	64 s	–	–	320 s
	Communication	$\leq$ 100 KB	$\approx$ 200 KB	–	–	4.2 GB
	Prover memory	$\approx$ 86 GB	$\approx$ 18 GB	–	–	$\leq$ 400 MB

TABLE I: **Comparing our ZK protocol with prior work.** The first example proves knowledge of 256 leaves that hash to a public root of a Merkle tree based on SHA-256 (511 hash-function evaluations). The second example proves knowledge of two 512×512 matrices over a 61-bit field whose product is a public matrix (roughly 134 million field multiplications). Performance of our protocol ($\rho = 40$, $\kappa = 128$) is measured by running the prover and verifier on two machines, each using 1 thread, connected via a 200 Mbps network, and is the total running time of both the offline and online phases. For ZKGC and Wolverine, the prover and verifier can execute the protocol in a *pipelined* fashion, which is why the overall time is the maximum of the prover and verifier times. Spartan uses a 256-bit field while Virgo and Wolverine use a 61-bit field. See Section V for details.

Online phase. The online phase of our protocol can be viewed as adapting the core idea of the ZKGC approach by viewing a ZK proof as a special case of secure two-party computation (2PC) where one party has no input. We differ from the ZKGC approach in the underlying 2PC protocol we use as our starting point: rather than using garbled circuits, we instead rely on a "GMW-style" approach [36] using authenticated triples [5], [51] (whose values are known to the prover) generated during the offline phase. A drawback of GMW-style protocols in the context of generic 2PC is that they have round complexity linear in the depth of the circuit being evaluated. Crucially, in the ZK context, we can exploit the fact that only one party has input to obtain an online phase that runs in constant rounds (or can even be non-interactive in the random-oracle model).

The prover and verifier run in linear time since they each make only one pass over the circuit. Moreover, they can evaluate the circuit "on-the-fly" (i.e., with memory overhead linear in what is needed to evaluate the circuit non-cryptographically), which allows our protocol to scale easily to very large circuits. Our approach is communication-efficient as well: for a circuit with C multiplication gates over an arbitrary finite field $\mathbb{F}_p$, the marginal communication complexity is only either $3\rho/\log C + 1$ elements per gate for small fields or 2–4 elements per gate for large fields.

Instantiating the offline phase. During the offline phase we set up authenticated multiplication triples (over the relevant field $\mathbb{F}_p$) between the prover and verifier using subfield Vector Oblivious Linear Evaluation (sVOLE) [13], [15]. For boolean circuits (i.e., $p = 2$), we use the recent work by Yang et al. [60] to generate an initial pool of authenticated bits, and then use those authenticated bits to generate authenticated triples as in prior work [52]. For $p > 2$, we extend the protocol of Yang et al. to obtain an efficient sVOLE protocol for arbitrary fields (which we believe to be of independent interest). We defer further details to Section IV.

B. Performance and Comparison to Prior Work

We have implemented Wolverine for both boolean and arithmetic circuits. Running over a 200 Mbps network, Wolverine processes boolean circuits at the rate of 2,000,000 AND gates per second (XOR gates are free), and arithmetic circuits over a 61-bit large field at the rate of 600,000 multiplication gates per second (addition gates are free). In Table I we provide benchmarks comparing Wolverine to prior work for two examples: proving knowledge of the leaves that hash to a Merkle-tree root (naturally represented as a boolean circuit) and proving knowledge of the inputs to matrix multiplication over a large field (naturally represented as an arithmetic circuit). In the boolean setting, Wolverine uses $15\times$ less communication than ZKGC [42] along with lower running time; Wolverine outperforms all other work in terms of overall time and memory usage. In the arithmetic setting, Wolverine is $5\times$ slower than Virgo [63] but needs only 3% of the memory. The advantage in memory usage would be even larger for larger circuits, and would enable Wolverine to scale to circuits larger than what can be feasibly handled by Virgo.

Comparison to ZK proofs based on VOLE/OT. Boyle et al. [13], [15] also proposed a framework for ZK proofs in which an offline phase is used to set up correlated randomness between the prover and verifier, and the subsequent online phase is non-interactive. With regard to the online phase, the primary advantages of their work are that the online phase can be non-interactive without the random-oracle model, and can be run any polynomial number of times following a single execution of the offline phase (that is, the offline phase is *reusable*). An advantage of our work is that it applies to circuits over arbitrary fields, whereas the work of Boyle et al. applies either to boolean circuits [15] or arithmetic circuits over large fields [13]. More to the point, the focus of our work is concrete efficiency, which was not investigated by Boyle et al. For boolean circuits, the ZK protocol of Boyle et al. [15] based on oblivious transfer requires communicating over 100,000 bits per gate when $\rho = 40$, which is four orders of magnitude larger than our protocol. For large fields, the VOLE-based ZK protocol of Boyle et al. [13] requires communication of at least 16 elements per gate, whereas our protocol sends only 2–4 elements per gate. We also offer concrete efficiency improvements for the offline phase in the large-field case. In particular, our sVOLE protocol avoids the generic, maliciously secure two-party computation used by Boyle et al. [13].

Comparison to zk-SNARKs. Our ZK protocol occupies a different portion of the solution space than (existing) zk-SNARKs. Existing zk-SNARKs impose concretely high memory requirements on the prover (cf. Table I), even when the memory requirements are linear in the circuit size. (While there are zk-SNARKs in which the prover asymptotically uses sublinear memory [24], such schemes are currently $\approx 200\times$ slower than state-of-the-art zk-SNARKs that uses linear memory [54].) The prover memory in Wolverine is significantly lower, allowing Wolverine to scale to very large circuits. On the other hand, zk-SNARKs have many advantages: they are non-interactive and have lower communication. They also have better efficiency for the verifier, although their *overall* time (i.e., including the time for the prover to generate the proof) might be longer.

In independent and concurrent work, Dittmer, Ishai, and Ostrovsky [29] have also developed a ZK protocol based on VOLE. They focus on communication complexity rather than concrete performance; their protocol only considers the case of large fields, and has lower communication complexity than our protocol in that case. Subsequent to our work, Baum, Malozemoff, Rosen and Scholl [3] have also proposed a different VOLE-based ZK protocol.

Organization of the paper. After reviewing some preliminaries in Section II, we describe the online phase of our ZK proof in Section III. In Section IV we describe the details of our sVOLE construction used in the offline phase of our ZK proof. We provide experimental results in Section V.

II. Preliminaries

We use κ and ρ to denote the computational and statistical security parameters, respectively. We let $\mathsf{negl}(\cdot)$ denote a negligible function, and use $\log$ to denote logarithms in base 2. We write $x \leftarrow S$ to denote sampling x uniformly from a set S, and $x \leftarrow \mathcal{D}$ to denote sampling x according to a distribution $\mathcal{D}$. We define $[a, b) = \{a, \ldots, b-1\}$ and write $[n] = \{1, \ldots, n\}$. We use bold lower-case letters like $\boldsymbol{a}$ for row vectors, and bold upper-case letters like $\mathbf{A}$ for matrices. We let $\boldsymbol{a}[i]$ denote the ith component of $\boldsymbol{a}$ (with $\boldsymbol{a}[0]$ the first entry), and let $\boldsymbol{a}[i:j)$ represent the subvector $(\boldsymbol{a}[i], \ldots, \boldsymbol{a}[j-1])$.

A circuit $\mathcal{C}$ over a field $\mathbb{F}_p$ is defined by a set of input wires $\mathcal{I}_{\mathsf{in}}$ and output wires $\mathcal{I}_{\mathsf{out}}$, along with a list of gates of the form $(\alpha, \beta, \gamma, T)$, where α, β are the indices of the input wires of the gate, γ is the index of the output wire of the gate, and $T \in \{\mathsf{Add}, \mathsf{Mult}\}$ is the type of the gate. If $p = 2$, then $\mathcal{C}$ is a boolean circuit with $\mathsf{Add} = \oplus$ and $\mathsf{Mult} = \wedge$. If $p > 2$ is prime, then $\mathcal{C}$ is an arithmetic circuit where Add/Mult correspond to addition/multiplication in $\mathbb{F}_p$. We let C denote the number of Mult gates in the circuit.

When we work in an extension field $\mathbb{F}_{p^r}$ of $\mathbb{F}_p$, we fix some monic, irreducible polynomial $f(X)$ of degree r and so $\mathbb{F}_{p^r} \cong \mathbb{F}_p[X]/f(X)$. We let $\mathsf{X} \in \mathbb{F}_{p^r}$ denote the element corresponding to $X \in \mathbb{F}_p[X]/f(X)$; thus, every $w \in \mathbb{F}_{p^r}$ can be written uniquely as $w = \sum_{i=0}^{r-1} w_i \cdot \mathsf{X}^i$ with $w_i \in \mathbb{F}_p$ for all i, and we may view elements of $\mathbb{F}_{p^r}$ equivalently as vectors in $\mathbb{F}_p^r$. When we write arithmetic expressions involving both elements of $\mathbb{F}_p$ and elements of $\mathbb{F}_{p^r}$, it is understood that values in $\mathbb{F}_p$ are viewed as lying in $\mathbb{F}_{p^r}$ in the natural way. We let $\mathbb{F}^*$ denote the nonzero elements of a field $\mathbb{F}$.

A. Information-Theoretic MACs and Batch Opening

We use *information-theoretic message authentication codes* (IT-MACs) [51], [26] to authenticate values in a finite field $\mathbb{F}_p$ using an extension field $\mathbb{F}_{p^r} \supseteq \mathbb{F}_p$. In more detail, let $\Delta \in \mathbb{F}_{p^r}$ be a *global key*, sampled uniformly, that is known only by one party $\mathsf{P_B}$. A value $x \in \mathbb{F}_p$ known by the other party $\mathsf{P_A}$ can be authenticated by giving $\mathsf{P_B}$ a uniform key $\mathsf{K}[x] \in \mathbb{F}_{p^r}$ and giving $\mathsf{P_A}$ the corresponding MAC tag

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

We denote such an authenticated value by $[x]$. Authenticated values are additively homomorphic, i.e., if $\mathsf{P_A}$ and $\mathsf{P_B}$ hold authenticated values $[x], [x']$ then they can locally compute $[x''] = [x + x']$ by having $\mathsf{P_A}$ set $x'' := x + x'$ and $\mathsf{M}[x''] := \mathsf{M}[x] + \mathsf{M}[x']$ and having $\mathsf{P_B}$ set $\mathsf{K}[x''] := \mathsf{K}[x] + \mathsf{K}[x']$. Similarly, for a public value $b \in \mathbb{F}_p$, the parties can locally compute $[y] = [x+b]$ or $[z] = [bx]$. We denote these operations by $[x''] = [x] + [x']$, $[y] = [x] + b$, and $[z] = b \cdot [x]$, respectively.

We extend the above notation to vectors of authenticated values as well. In that case, $[\boldsymbol{u}]$ means that (for some n) $\mathsf{P_A}$ holds $\boldsymbol{u} \in \mathbb{F}_p^n$ and $\boldsymbol{w} \in \mathbb{F}_{p^r}^n$, while $\mathsf{P_B}$ holds $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ with $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$. An *authenticated multiplication triple* consists of authenticated values $[x], [y], [z]$ where $z = x \cdot y$.

Batch opening of authenticated values. An authenticated value $[x]$ can be "opened" by having $\mathsf{P_A}$ send $x \in \mathbb{F}_p$ and $\mathsf{M}[x] \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$, who then verifies that $\mathsf{M}[x] \overset{?}{=} \mathsf{K}[x] + \Delta \cdot x$. This has soundness error $1/p^r$, and requires sending an additional $r \log p$ bits (beyond x itself). While this can be repeated in parallel when opening multiple authenticated values $[x_1], \ldots, [x_\ell]$, communication can be reduced using batching [51], [26]. We describe two approaches in Appendix B. Hereafter, we write $\mathsf{Open}([\boldsymbol{x}])$ to denote a generic batch opening of a vector of authenticated values. In addition, we write $\mathsf{CheckZero}([\boldsymbol{x}])$ for the special case where all x_i are supposed to be 0 and so need not be sent. We let $\varepsilon_{\mathsf{open}}$ denote the soundness error (which depends on the technique used); when using either of the techniques described above, $\varepsilon_{\mathsf{open}}$ is independent of the number ℓ of authenticated values opened.

B. Security Model and Functionalities

We use the *universal composability* (UC) framework [22] to prove security in the presence of a malicious, static adversary.

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

Upon receiving $(\mathsf{prove}, \mathcal{C}, w)$ from a prover $\mathcal{P}$ and $(\mathsf{verify}, \mathcal{C})$ from a verifier $\mathcal{V}$ where the same (boolean or arithmetic) circuit $\mathcal{C}$ is input by both parties, send true to $\mathcal{V}$ if $\mathcal{C}(w) = 1$; otherwise, send false to $\mathcal{V}$.

Fig. 1: **The zero-knowledge functionality.**

Functionality $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$

Initialize: Upon receiving init from $\mathsf{P_A}$ and $\mathsf{P_B}$, sample $\Delta \leftarrow \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest or receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ and send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands.

Extend: This procedure can be run multiple times. Upon receiving (extend, ℓ) from $\mathsf{P_A}$ and $\mathsf{P_B}$, do:

1) If $\mathsf{P_B}$ is honest, sample $\mathsf{K}[\boldsymbol{x}] \leftarrow \mathbb{F}_{p^r}^{\ell}$. Otherwise, receive $\mathsf{K}[\boldsymbol{x}] \in \mathbb{F}_{p^r}^{\ell}$ from the adversary.
2) If $\mathsf{P_A}$ is honest, sample $\boldsymbol{x} \leftarrow \mathbb{F}_p^{\ell}$ and compute $\mathsf{M}[\boldsymbol{x}] := \mathsf{K}[\boldsymbol{x}] + \Delta \cdot \boldsymbol{x} \in \mathbb{F}_{p^r}^{\ell}$. Otherwise, receive $\boldsymbol{x} \in \mathbb{F}_p^{\ell}$ and $\mathsf{M}[\boldsymbol{x}] \in \mathbb{F}_{p^r}^{\ell}$ from the adversary, and then recompute $\mathsf{K}[\boldsymbol{x}] := \mathsf{M}[\boldsymbol{x}] - \Delta \cdot \boldsymbol{x} \in \mathbb{F}_{p^r}^{\ell}$.
3) Send $(\boldsymbol{x}, \mathsf{M}[\boldsymbol{x}])$ to $\mathsf{P_A}$ and $\mathsf{K}[\boldsymbol{x}]$ to $\mathsf{P_B}$.

Global-key query: If $\mathsf{P_A}$ is corrupted, receive $(\mathsf{guess}, \Delta')$ from the adversary with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send success to $\mathsf{P_A}$ and ignore any subsequent global-key query. Otherwise, send abort to both parties and abort.

Fig. 2: **Functionality for subfield VOLE.**

We say that a protocol Π *UC-realizes* an ideal functionality $\mathcal{F}$ if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT adversary (simulator) $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$ with arbitrary auxiliary input z, the output distribution of $\mathcal{Z}$ in the *real-world* execution where the parties interact with $\mathcal{A}$ and execute Π is computationally indistinguishable from the output distribution of $\mathcal{Z}$ in the *ideal-world* execution where the parties interact with $\mathcal{S}$ and $\mathcal{F}$.

The protocol that we construct in this work UC-realizes the standard zero-knowledge functionality $\mathcal{F}_{\mathsf{ZK}}$, reproduced in Figure 1 for completeness. (We omit session identifiers in all our ideal functionalities for the sake of readability.) Our ZK protocol relies on the *subfield Vector Oblivious Linear Evaluation* (sVOLE) functionality (see Figure 2), which is the same as that by Boyle et al. [14], except that the adversary is allowed to make a global-key query on Δ and would incur aborting for an incorrect guess. After an initialization that is done once, this functionality allows two parties to repeatedly generate a vector of authenticated values known to $\mathsf{P_A}$. Other functionalities are given for reference in Appendix A.

III. Our Zero-Knowledge Protocol

In Figure 3, we describe our zero-knowledge protocol Π_{ZK}, which operates in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model. As noted in Section I-A, our protocol can be viewed as following a "GMW-style" approach to secure two-party computation using authenticated multiplication triples [51], [26]. In the secure-computation setting, the evaluation of a multiplication gate

Protocol Π_{ZK}

Inputs and parameters: The prover $\mathcal{P}$ and verifier $\mathcal{V}$ hold a circuit $\mathcal{C}$ over a finite field $\mathbb{F}_p$ with C multiplication gates; $\mathcal{P}$ holds a witness w such that $\mathcal{C}(w) = 1$. Fix parameters B, c, and r, and let $\ell = C \cdot B + c$.

Offline phase:

1) $\mathcal{P}$ (acting as $\mathsf{P_A}$) and $\mathcal{V}$ (acting as $\mathsf{P_B}$) send init to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.
2) $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{extend}, |\mathcal{I}_{\mathsf{in}}| + 3\ell + C)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, which returns authenticated values $\{[\lambda_i]\}_{i \in \mathcal{I}_{\mathsf{in}}}$, $\{([x_i], [y_i], [r_i])\}_{i \in [\ell]}$, and $\{[s_i]\}_{i \in [C]}$ to the parties.
(If $\mathcal{V}$ receives abort from $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, then it aborts.)
3) For $i \in [\ell]$, $\mathcal{P}$ sends $d_i := x_i \cdot y_i - r_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[z_i] := [r_i] + d_i$.

Online phase:

4) For $i \in \mathcal{I}_{\mathsf{in}}$, $\mathcal{P}$ sends $\Lambda_i := w_i - \lambda_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_i] := [\lambda_i] + \Lambda_i$.
5) For each gate $(\alpha, \beta, \gamma, T) \in \mathcal{C}$, in topological order:
   a) If $T = \mathsf{Add}$, then the two parties locally compute $[w_\gamma] := [w_\alpha] + [w_\beta]$.
   b) If $T = \mathsf{Mult}$ and this is the ith multiplication gate, $\mathcal{P}$ sends $d := w_\alpha \cdot w_\beta - s_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_\gamma] := [s_i] + d$.
6) $\mathcal{V}$ samples a random permutation π on $\{1, \ldots, \ell\}$ and sends it to $\mathcal{P}$. The two parties use π to permute the $\{([x_i], [y_i], [z_i])\}_{i \in [\ell]}$ obtained in step 3.
7) For the ith multiplication gate $(\alpha, \beta, \gamma, \mathsf{Mult})$, where the parties obtained $([w_\alpha], [w_\beta], [w_\gamma])$ in step 5, do the following for $j = 1, \ldots, B$:
   a) Let $([x], [y], [z])$ be the $\big((i-1)B + j\big)$th authenticated triple (after applying π in step 6).
   b) The parties run $\delta_\alpha := \mathsf{Open}([w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$. The parties then compute $[\mu] := [z] - [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta$, and finally run $\mathsf{CheckZero}([\mu])$.
8) For each of the remaining c authenticated triples, say $([x], [y], [z])$, the parties run $x := \mathsf{Open}([x])$ and $y := \mathsf{Open}([y])$. They also compute $[\nu] := [z] - x \cdot y$ and then run $\mathsf{CheckZero}([\nu])$.
9) For the single output wire $o \in \mathcal{I}_{\mathsf{out}}$ with authenticated value $[w_o]$, the parties run $\mathsf{CheckZero}([w_o] - 1)$.

Fig. 3: **Zero-knowledge proof in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model.**

requires two rounds of interaction, since the parties hold shares of the values on the input wires, but neither party knows those values. In the ZK setting, however, the prover $\mathcal{P}$ knows the values on all wires; thus, evaluation of a multiplication gate can be done without any interaction at all.

At a high level, our protocol consists of the following steps:

1) **Initialization.** The parties prepare authenticated values $\{[\lambda_i]\}$ for the witness, and $\{[s_i]\}$ for each multiplication gate in the circuit. The parties also generate some number of authenticated multiplication triples $\{([x_i], [y_i], [z_i])\}$; a malicious prover may cause some or all of these triples to be incorrect (i.e., $z_i \neq x_i \cdot y_i$).
2) **Circuit evaluation.** Starting with the authenticated values $\{[w_i]\}$ at the input wires, the parties inductively compute authenticated values for all the wires in the circuit. For addition gates, this is easy. For the i-th multiplication gate,

the prover uses $[s_i]$ to enable the verifier to compute its component of the authenticated value for the output wire without revealing information about the values on the input wires. Specifically, given authenticated values $[w_\alpha], [w_\beta]$ on the input wires to the ith multiplication gate, the prover sends $w_\alpha \cdot w_\beta - s_i$ to the verifier; the prover and verifier then compute $[w_\gamma] := [s_i] + (w_\alpha \cdot w_\beta - s_i)$ as the authenticated value of the output wire. All communication here is from the prover to the verifier, so the entire circuit can be evaluated using only one round of communication.

Once the parties have an authenticated value $[w_o]$ for the output wire, the prover simply opens that value, and the verifier checks that it is equal to 1.

3) **Verifying correct behavior.** So far, nothing prevents a malicious prover from cheating. To detect cheating, the verifier needs to check the behavior of the prover at each multiplication gate using the initial set of authenticated multiplication triples the parties generated. This can be done in various ways. In the protocol as described in Figure 3, which works for circuits over an arbitrary field, the verifier checks the behavior of the prover as follows (adapting [2]):
   - The verifier checks a random subset of the authenticated triples to make sure they are correctly formed. For an authenticated multiplication triple $([x], [y], [z])$, this can be done by having the prover run $\mathsf{Open}([x])$ and $\mathsf{Open}([y])$ followed by $\mathsf{CheckZero}([z] - x \cdot y)$.
   - The verifier then uses the remaining authenticated triples to check that each multiplication gate was computed correctly. For a multiplication gate with authenticated values $[w_\alpha], [w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire, the relation $w_\gamma = w_\alpha w_\beta$ can be checked using an authenticated multiplication triple $([x], [y], [z])$ by having the prover run $\delta_\alpha := \mathsf{Open}([w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$, followed by
$$\mathsf{CheckZero}\Big([z] - [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta\Big).$$
     Each multiplication gate is checked in this way using B authenticated multiplication triples.

   In Section III-B, we describe other approaches for verifying correct behavior.

Note that the checks for the openings of all the authenticated values (i.e., all the executions of Open and CheckZero) can be batched together at the end of the protocol.

Non-interactive online phase. The ZK protocol described in Figure 3 can be implemented in constant rounds. If we use the Fiat-Shamir heuristic both for deriving the permutation π as well as for non-interactive opening of authenticated values, the online phase can be made non-interactive.

A. *Proof of Security*

Before giving the proof of security for Π_{ZK}, we analyze the procedure used to check correctness of the multiplication gates. Consider some multiplication gate with authenticated values $[w_\alpha]$, $[w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire. If $\mathcal{P}$ cheated, so $w_\gamma \neq w_\alpha \cdot w_\beta$, then this cheating will be detected in step 7 of the protocol unless all B of the multiplication triples used to check that gate are incorrect. (We ignore for now the possibility that $\mathcal{P}$ is able to successfully cheat when running Open/CheckZero.) But if too many of the initial multiplication triples are incorrect, then there is a high probability that $\mathcal{P}$ will be caught in step 8. We can analyze the overall probability with which a cheating $\mathcal{P}$ can successfully evade detection by considering an abstract "balls-and-bins" game with an adversary $\mathcal{A}$, which is based on a similar game considered previously in the context of secure three-party computation [2]. The game proceeds as follows:

1) $\mathcal{A}$ prepares $\ell = CB + c$ balls $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$, each of which is either good or bad. $\mathcal{A}$ also prepares C bins, each of which is either good or bad. The balls $\{\mathcal{B}_i\}_{i\in[\ell]}$ correspond to the triples $\{([x_i], [y_i], [z_i])\}_{i\in[\ell]}$ defined in step 3 of the protocol, and the bins correspond to the triples $\{([w_\alpha], [w_\beta], [w_\gamma])\}$ defined for the multiplication gates during the circuit evaluation.
2) Then, c random balls are chosen. If any of the chosen balls is bad, $\mathcal{A}$ loses. Otherwise, the game continues to proceed.
3) The remaining CB balls are randomly partitioned into the C bins, with each bin receiving exactly B balls.
4) We say that a bin is fully good (resp., fully bad) if it is labeled good and all the balls inside it are good (resp., labeled bad and all the balls inside it are bad). $\mathcal{A}$ wins if and only if there exists at least one bin that is fully bad, and all other bins are either fully good or fully bad.

Lemma 1. *Assume $c \geq B$. Then $\mathcal{A}$ wins the above game with probability at most* $\binom{CB+c}{B}^{-1}$.

Proof. Assume $\mathcal{A}$ makes m bins bad for $1 \leq m \leq C$. It is easy to see that $\mathcal{A}$ can only possibly win if exactly mB balls among $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$ are bad, and they are exactly placed in the m bins that are bad. We compute the probability that $\mathcal{A}$ wins for some fixed m.

Since exactly mB balls of the $\ell = CB + c$ balls are bad, the probability that none of the bad balls is chosen in step 2 of the game is exactly
$$\frac{\binom{\ell-mB}{c}}{\binom{\ell}{c}} = \frac{(\ell-mB)! \cdot (\ell-c)!}{\ell! \cdot (\ell-mB-c)!} = \frac{(CB+c-mB)! \cdot (CB)!}{(CB+c)! \cdot (CB-mB)!}.$$
Assume that this occurs. We are left with $\ell - c = CB$ balls, of which mB are bad. The probability that B bad balls are placed in each bad bin is
$$p_1 = \frac{(mB)! \cdot (CB-mB)!}{(CB)!}.$$
Thus, the probability that $\mathcal{A}$ wins is exactly
$$\frac{\binom{\ell-mB}{c}}{\binom{\ell}{c}} \cdot p_1 = \frac{(CB+c-mB)! \cdot (mB)!}{(CB+c)!} = \binom{CB+c}{mB}^{-1}.$$
For $c \geq B$, $1 \leq m \leq C$, this is maximized when $m = 1$. □

Now we prove security of protocol Π_{ZK}.

Theorem 1. *Let $c \geq B$. Protocol Π_{ZK} UC-realizes $\mathcal{F}_{\mathsf{ZK}}$ in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model. In particular, no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability $\leq \binom{CB+c}{B}^{-1} + p^{-r} + \varepsilon_{\mathsf{open}}$.*

Proof. We first consider the case of a malicious prover (i.e., soundness) and then consider the case of a malicious verifier (i.e., zero knowledge). In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{ZK}}$, and running the PPT adversary $\mathcal{A}$ as a subroutine while emulating functionality $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ for $\mathcal{A}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and $\mathcal{Z}$.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ for $\mathcal{A}$ by choosing uniform $\Delta \in \mathbb{F}_{p^r}$ and recording all the values $\{\lambda_i\}_{i \in \mathcal{I}_{\mathsf{in}}}$, $\{(x_i, y_i, r_i)\}_{i \in [\ell]}$, and $\{s_i\}_{i \in [C]}$, and their corresponding MAC tags, sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$. These values define corresponding keys in the natural way.
2) If $\mathcal{A}$ makes a global-key query $(\mathsf{guess}, \Delta')$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, then $\mathcal{S}$ checks if $\Delta = \Delta'$. If not, $\mathcal{S}$ sends abort to $\mathcal{A}$, sends $(\mathsf{prove}, \mathcal{C}, \perp)$ to $\mathcal{F}_{\mathsf{ZK}}$, and aborts. Otherwise, $\mathcal{S}$ sends success to $\mathcal{A}$ and continues.
3) When $\mathcal{A}$ sends $\{\Lambda_i\}_{i \in \mathcal{I}_{\mathsf{in}}}$ in step 4, $\mathcal{S}$ sets $w_i := \lambda_i + \Lambda_i$ for $i \in \mathcal{I}_{\mathsf{in}}$.
4) $\mathcal{S}$ runs the rest of the protocol as an honest verifier, using Δ and the keys defined in the first step. If the honest verifier outputs false, then $\mathcal{S}$ sends $(\mathsf{prove}, \mathcal{C}, \perp)$ to $\mathcal{F}_{\mathsf{ZK}}$ and aborts. If the honest verifier outputs true, then $\mathcal{S}$ sends $(\mathsf{prove}, \mathcal{C}, w)$ to $\mathcal{F}_{\mathsf{ZK}}$ where w is defined as above.

We assume that $\mathcal{A}$ does not correctly guess Δ; this is true except with probability at most p^{-r}. It is clear that the view of $\mathcal{A}$ is perfectly simulated by $\mathcal{S}$. Whenever the verifier simulated by $\mathcal{S}$ outputs false, the real verifier outputs false as well (since $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{ZK}}$). It thus only remains to bound the probability with which the simulated verifier run by $\mathcal{S}$ outputs true but the witness w sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{ZK}}$ satisfies $\mathcal{C}(w) = 0$. Below, we show that if $\mathcal{C}(w) = 0$ then the probability that the simulated verifier outputs true is at most $\binom{CB+c}{B}^{-1} + \varepsilon_{\mathsf{open}}$.

If $\mathcal{C}(w) = 0$ then either $w_o = 0$ or else at least one of the triples $\{([w_\alpha], [w_\beta], [w_\gamma])\}$ defined at the multiplication gates during the circuit evaluation must be incorrect. In the former case, the probability that $\mathcal{P}$ succeeds when running $\mathsf{CheckZero}([w_o] - 1)$ is at most $\varepsilon_{\mathsf{open}}$. In the latter case, Lemma 1 shows that the probability that $\mathcal{A}$ avoids being "caught" in steps 6–8 is at most $\binom{CB+c}{B}^{-1}$; if $\mathcal{A}$ is caught, then it succeeds in opening some incorrect value with probability at most $\varepsilon_{\mathsf{open}}$. This completes the proof for the case of a malicious prover.

Malicious verifier. If $\mathcal{S}$ receives false from $\mathcal{F}_{\mathsf{ZK}}$, then it simply aborts. Otherwise, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by recording the global key Δ, and the keys for all the authenticated values, sent to the functionality by $\mathcal{A}$. Then, $\mathcal{S}$ samples uniform values for $\{\lambda_i\}_{i \in \mathcal{I}_{\mathsf{in}}}$, $\{(x_i, y_i, r_i)\}_{i \in [\ell]}$, and $\{s_i\}_{i \in [C]}$, and computes their corresponding MAC tags in the natural way.
2) $\mathcal{S}$ executes steps 3–8 of protocol Π_{ZK} by simulating the honest prover with input $w = 0^{|\mathcal{I}_{\mathsf{in}}|}$.
3) In step 9, $\mathcal{S}$ computes $\mathsf{K}[w_o]$ (based on the keys sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$) and then sets $\mathsf{M}[w_o] := \mathsf{K}[w_o] + \Delta$. Finally, it uses $\mathsf{M}[w_o]$ to run $\mathsf{CheckZero}([w_o] - 1)$ with $\mathcal{A}$.

The view of $\mathcal{A}$ simulated by $\mathcal{S}$ is distributed identically to its view in the real protocol execution. □

B. Other Approaches for Verifying Correct Behavior

Here we describe alternative approaches for checking correctness of multiplication gates for large p (i.e., $\log p \geq \rho$).

Approach 1. The first approach can be viewed as a simplified version of the check used by SPDZ [26]. Both parties now prepare a *single* authenticated multiplication triple $([x], [y], [z])$ per multiplication gate (so only C in total), which may be incorrect if $\mathcal{P}$ is malicious. To check correctness of a multiplication gate with authenticated values $[w_\alpha]$, $[w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire, the verifier sends a uniform $\eta \in \mathbb{F}_p$ to the prover, who responds by running $\delta_\alpha := \mathsf{Open}(\eta \cdot [w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$, followed by

$$\mathsf{CheckZero}([z] - \eta \cdot [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta).$$

This has soundness error $1/p + \varepsilon_{\mathsf{open}}$. To see this, say $w_\gamma = w_\alpha w_\beta + \Delta_w$ with $\Delta_w \neq 0$, and let $z = xy + \Delta_z$. Then $z - \eta \cdot w_\gamma + \delta_\beta \cdot x + \delta_\alpha \cdot y + \delta_\alpha \cdot \delta_\beta = 0$ iff $\eta = \Delta_z / \Delta_w$, which occurs with probability $1/p$. Note that this checking procedure can be done for all multiplication gates in parallel using a single value η, and the overall soundness error remains unchanged. It can also be made non-interactive using the Fiat-Shamir heuristic in the random-oracle model.

Approach 2: Trading off communication and computation. This approach, which is a simplified and improved variant of the polynomial approach used by SPDZ [26], reduces the communication complexity by roughly half (from 4 to 2 field elements per gate) at the expense of increased computation. Intuitively, the prover and verifier define polynomials F, G, H that interpolate to $\{w_\alpha^i\}$, $\{w_\beta^i\}$, and $\{w_\gamma^i\}$, respectively. If $w_\gamma^i = w_\alpha^i \cdot w_\beta^i$ for all i, then $H = F \cdot G$, and this can be verified by checking whether $H(\nu) = F(\nu) \cdot G(\nu)$ at a random point $\nu \in \mathbb{F}_{p^r}$. Details follow.

Assume $p \geq 2C - 1$. Let $([w_\alpha^i], [w_\beta^i], [w_\gamma^i])$ be the authenticated values corresponding to the ith multiplication gate. The parties additionally compute $C - 1$ authenticated values $\{[s_i]\}_{i \in [C+1, 2C)}$; they also compute an authenticated multiplication triple $([x], [y], [z])$ (which may be incorrect if $\mathcal{P}$ is malicious) with $x, y, z \in \mathbb{F}_{p^r}$.[1] They then do the following:

1) Let $F \in \mathbb{F}_p[X]$ (resp., $G \in \mathbb{F}_p[X]$) be the polynomial of degree at most $C - 1$ such that $F(i) = w_\alpha^i$ (resp., $G(i) = w_\beta^i$) for $i \in [C]$. Note that $\mathcal{P}$ can compute F and G explicitly, and $\mathcal{P}$ and $\mathcal{V}$ can compute the authenticated

[1]A uniform authenticated value $[z]$ with $z \in \mathbb{F}_{p^r}$ can be generated from r uniform authenticated values $[z_1], \ldots, [z_r]$ with $z_i \in \mathbb{F}_p$ by setting $z = \sum_i z_i \cdot \mathsf{X}^i$. An authenticated triple can be computed from such authenticated values in the natural way.

value $[w_\alpha^k] \overset{\text{def}}{=} [F(k)]$ (resp., $[w_\beta^k] \overset{\text{def}}{=} [G(k)]$) for any $k \in \mathbb{F}_{p^r}$ using Lagrange interpolation over the shares $\{[w_\alpha^i]\}_{i\in[C]}$ (resp., $\{[w_\beta^i]\}_{i\in[C]}$).

2) For $k \in [C+1, 2C)$, $\mathcal{P}$ sends $d'_k := w_\alpha^k \cdot w_\beta^k - s_k$ to $\mathcal{V}$, and both parties compute $[w_\gamma^k] := [s_k] + d'_k$. Let $H \in \mathbb{F}_p[X]$ be the polynomial of degree at most $2C-2$ such that $H(i) = w_\gamma^i$ for $i \in [2C-1]$. Note that $\mathcal{P}$ can compute H explicitly, while $\mathcal{P}$ and $\mathcal{V}$ can compute the authenticated value $[H(k)]$ for any $k \in \mathbb{F}_{p^r}$ using Lagrange interpolation over the shares $[w_\gamma^i]$.
3) $\mathcal{V}$ sends a uniform $\nu \in \mathbb{F}_{p^r}$ to $\mathcal{P}$. Then the parties compute authenticated values $[F(\nu)]$, $[G(\nu)]$, and $[H(\nu)]$.
4) Finally, $\mathcal{V}$ verifies that $F(\nu)\cdot G(\nu) = H(\nu)$ as in approach 1, above. That is, $\mathcal{V}$ sends a uniform $\eta \in \mathbb{F}_{p^r}$ to $\mathcal{P}$, who responds by running $\delta := \mathsf{Open}(\eta \cdot [F(\nu)] - [x])$ and $\sigma := \mathsf{Open}([G(\nu)] - [y])$, followed by

$$\mathsf{CheckZero}([z] - \eta\cdot[H(\nu)] + \sigma\cdot[x] + \delta\cdot[y] + \delta\cdot\sigma).$$

This has soundness error $(2C-1)/p^r + \varepsilon_{\mathsf{open}}$. To see this, note that if there exists an $i \in [C]$ with $w_\alpha^i \cdot w_\beta^i \neq w_\gamma^i$ then the polynomials $F\cdot G$ and H are different, and so agree in at most $2C-2$ points. Thus, $F(\nu)\cdot G(\nu) \neq H(\nu)$ except with probability at most $(2C-2)/p^r$. When that is the case, an analysis in the first approach shows that the final check fails except with probability at most $1/p^r + \varepsilon_{\mathsf{open}}$.

This approach can also be made non-interactive using the Fiat-Shamir heuristic in the random-oracle model.

IV. Subfield VOLE

In this section, we present an sVOLE protocol that can be used during the offline phase of our ZK protocol. In Section IV-A, we first present an sVOLE protocol with linear communication complexity. Although this already suffices for our ZK protocol, we can obtain much better efficiency using “sVOLE extension” (by analogy with OT extension), by which we extend a small number of “base” sVOLE correlations into a larger number of sVOLE correlations. Toward this end, in Section IV-B we construct a protocol for *single-point* sVOLE (spsVOLE) in the $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$-hybrid model, where spsVOLE is like sVOLE except that the vector of authenticated values has only a *single* nonzero entry. Then, in Section IV-C, we present an efficient protocol for “sVOLE extension” using spsVOLE as a subroutine and relying on a variant of the *Learning Parity with Noise* (LPN) assumption. We provide some intuition for each protocol in the relevant section. Our implementation shows that this protocol outperforms all prior work; we discuss its concrete performance in Section V-A.

A. Base sVOLE Protocol

We present a “base” sVOLE protocol that is based on oblivious transfer (OT) and is inspired by prior work of Keller et al. [49], [50]. Our protocol relies on the *correlated oblivious product evaluation with errors* (COPEe) functionality $\mathcal{F}_{\mathsf{COPEe}}$, which extends the analogous functionality introduced by Keller et al. [50] to the subfield case we are interested in. We show in Appendix C how to UC-realize $\mathcal{F}_{\mathsf{COPEe}}$ from OT.

Functionality $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$

Initialize: Upon receiving init from parties $\mathsf{P_A}$, $\mathsf{P_B}$, sample $\Delta \leftarrow \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest, and receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ, send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands. Let $\boldsymbol{\Delta}_B \in \{0,1\}^{rm}$ be the bit-decomposition of Δ, where $m = \lceil \log p \rceil$.

Extend: Upon receiving (extend, u) with $u \in \mathbb{F}_p$ from $\mathsf{P_A}$ and (extend) from $\mathsf{P_B}$, this functionality operates as follows:
1) Sample $v \leftarrow \mathbb{F}_{p^r}$. If $\mathsf{P_B}$ is corrupted, instead receive $v \in \mathbb{F}_{p^r}$ from the adversary.
2) Compute $w := v + \Delta\cdot u \in \mathbb{F}_{p^r}$.
3) If $\mathsf{P_A}$ is corrupted, receive $w \in \mathbb{F}_{p^r}$ and $\boldsymbol{u} \in \mathbb{F}_p^{rm}$ from the adversary, and recompute
$$v := w - \langle \boldsymbol{g} * \boldsymbol{u}, \boldsymbol{\Delta}_B\rangle \in \mathbb{F}_{p^r},$$
where $*$ denotes the component-wise product.
4) Output (u, w) to $\mathsf{P_A}$ and v to $\mathsf{P_B}$.

Fig. 4: **COPEe functionality.**

Protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$

Sub-protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$ with selective-failure leakage:
1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$, which returns Δ to $\mathsf{P_B}$.
2) $\mathsf{P_A}$ samples $u_i \leftarrow \mathbb{F}_p$ for $i \in [0,n)$ and $a_h \leftarrow \mathbb{F}_p$ for $h \in [0,r)$. For $i \in [0,n)$, $\mathsf{P_A}$ sends (extend, u_i) to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ and $\mathsf{P_B}$ sends (extend) to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$, which returns $w_i \in \mathbb{F}_{p^r}$ to $\mathsf{P_A}$ and $v_i \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$ such that $w_i = v_i + \Delta\cdot u_i$. For $h \in [0,r)$, both parties also call $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ on respective inputs (extend, a_h) and (extend), following which $\mathsf{P_A}$ gets $c_h \in \mathbb{F}_{p^r}$ and $\mathsf{P_B}$ obtains $b_h \in \mathbb{F}_{p^r}$ such that $c_h = b_h + \Delta\cdot a_h$.
3) $\mathsf{P_B}$ samples $\chi_0, \ldots, \chi_{n-1} \leftarrow \mathbb{F}_{p^r}$, and sends them to $\mathsf{P_A}$. Then $\mathsf{P_A}$ computes $x := \sum_{i=0}^{n-1} \chi_i \cdot u_i + \sum_{h=0}^{r-1} a_h \cdot \mathsf{X}^h$, $z := \sum_{i=0}^{n-1} \chi_i \cdot w_i + \sum_{h=0}^{r-1} c_h \cdot \mathsf{X}^h$, and sends (x, z) to $\mathsf{P_B}$.
4) $\mathsf{P_B}$ computes $y := \sum_{i=0}^{n-1} \chi_i \cdot v_i + \sum_{h=0}^{r-1} b_h \cdot \mathsf{X}^h$ and checks that $z = y + \Delta\cdot x$. If not, $\mathsf{P_B}$ aborts.
5) For $i \in [0,n)$, $\mathsf{P_A}$ defines $\boldsymbol{u}[i] = u_i$ and $\boldsymbol{w}[i] = w_i$, and $\mathsf{P_B}$ sets $\boldsymbol{v}[i] = v_i$.

Full protocol without any leakage: Let $\ell = \lceil 2\rho/r \log p\rceil + 1$.
1) Both parties execute the above sub-protocol with parameters p and $k = \ell\cdot r$. Then, $\mathsf{P_A}$ obtains $(\boldsymbol{u}, \boldsymbol{w}) \in \mathbb{F}_p^n \times \mathbb{F}_{p^k}^n$ and $\mathsf{P_B}$ gets $\Delta \in \mathbb{F}_{p^k}$ and $\boldsymbol{v} \in \mathbb{F}_{p^k}^n$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta\cdot\boldsymbol{u}$. By viewing an element in $\mathbb{F}_{p^k}$ as a vector in $\mathbb{F}_{p^r}^\ell$, two parties obtain $\{\boldsymbol{w}_i\}_{i\in[\ell]}$ and $\{(\Delta_i, \boldsymbol{v}_i)\}_{i\in[\ell]}$ respectively such that $\boldsymbol{w}_i = \boldsymbol{v}_i + \boldsymbol{u}\cdot\Delta_i$, $\boldsymbol{w}_i, \boldsymbol{v}_i \in \mathbb{F}_{p^r}^n$ and $\Delta_i \in \mathbb{F}_{p^r}$.
2) $\mathsf{P_B}$ samples $\alpha_1, \ldots, \alpha_\ell \leftarrow \mathbb{F}_{p^r}$ and sends them to $\mathsf{P_A}$. $\mathsf{P_A}$ computes $\boldsymbol{t} := \sum_{i=1}^{\ell} \alpha_i \cdot \boldsymbol{w}_i$; $\mathsf{P_B}$ computes $\boldsymbol{s} := \sum_{i=1}^{\ell} \alpha_i \cdot \boldsymbol{v}_i$ and $\Gamma := \sum_{i=1}^{\ell} \alpha_i \cdot \Delta_i$, where $\boldsymbol{t} = \boldsymbol{s} + \Gamma\cdot\boldsymbol{u}$.
3) $\mathsf{P_A}$ outputs $\boldsymbol{u}$ and $\boldsymbol{t}$; $\mathsf{P_B}$ outputs Γ and $\boldsymbol{s}$.

Fig. 5: **Base sVOLE protocol in the $\mathcal{F}_{\mathsf{COPEe}}$-hybrid model.**

Functionality $\mathcal{F}_{\mathsf{COPEe}}$ is described in Figure 4, where $m = \lceil \log p \rceil$. In $\mathcal{F}_{\mathsf{COPEe}}$, we define a “gadget vector” $\boldsymbol{g} \in \mathbb{F}_{p^r}^{rm}$ by

$$\left((1,\ldots,2^{m-1}), (1,\ldots,2^{m-1})\cdot\mathsf{X}, \ldots, (1,\ldots,2^{m-1})\cdot\mathsf{X}^{r-1}\right).$$

For a vector $\boldsymbol{x} \in \mathbb{F}_{p^r}^{rm}$, we define

$$\langle \boldsymbol{g}, \boldsymbol{x}\rangle = \sum_{i=0}^{r-1}\Big(\sum_{j=0}^{m-1} \boldsymbol{x}[i\cdot m + j]\cdot 2^j\Big)\cdot \mathsf{X}^i \in \mathbb{F}_{p^r},$$

where the definition can be extended to the cases $\boldsymbol{x} \in \{0,1\}^{rm}$ or $\boldsymbol{x} \in \mathbb{F}_p^{rm}$ by viewing $\boldsymbol{x}$ as lying in $\mathbb{F}_{p^r}^{rm}$ in the natural way. The bit-decomposition of $\Delta \in \mathbb{F}_{p^r}$ is the string $\boldsymbol{\Delta}_B \in \{0,1\}^{rm}$ satisfying $\langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle = \Delta$.

In Figure 5, we present a protocol $\Pi^{p,r}_{\text{base-sVOLE}}$ that UC-realizes $\mathcal{F}^{p,r}_{\text{sVOLE}}$ in the $\mathcal{F}_{\text{COPEe}}$-hybrid model. We first describe a sub-protocol $\Pi^{p,r}_{\text{base-LsVOLE}}$, which allows two parties to generate sVOLE correlations with a selective-failure leakage on Δ, meaning that a malicious $\mathsf{P_A}$ is allowed to guess a subset of Δ and the protocol execution aborts for an incorrect guess. In this sub-protocol, $\mathsf{P_B}$ performs a *correlation check* in steps 3 and 4 to verify that the resulting sVOLE correlations are correct (i.e., $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$). Then, based on $\Pi^{p,r}_{\text{base-LsVOLE}}$, we show how to generate sVOLE correlations without such leakage using the leftover hash lemma [44]. In protocol $\Pi^{p,r}_{\text{base-sVOLE}}$, all the uniform coefficients (i.e., $\{\chi_i\}, \{\alpha_i\}$) can be computed from a random seed and a hash function modeled as a random oracle.

We prove the following in the full version of our work.

Theorem 2. *Protocol $\Pi^{p,r}_{\text{base-sVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\text{sVOLE}}$ in the $\mathcal{F}^{p,r}_{\text{COPEe}}$-hybrid model. In particular, no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution, except with probability at most $(r \log p)^2/p^r + 1/2^\rho$.*

Optimization. For many applications (e.g., our protocols) where learning the entire global key Δ is necessary in order to violate security of some higher-level protocol, it is unnecessary to eliminate the selective-failure leakage about Δ. This can be argued as follows. Assume the adversary guesses a set S (if there are multiple guesses then S is the intersection of all guessed sets) and is caught cheating if $\Delta \notin S$. The probability that the selective-failure attack is successful is $|S|/p^r$; conditioned on this event, the min-entropy of Δ is reduced to $\log|S|$. Therefore, the overall probability for the adversary to determine Δ is $|S|/p^r \cdot 2^{-\log|S|} = p^{-r}$, which is the same as the probability in the absence of any leakage. Similar observations have been used in secure-computation protocols [50], [25], [59].

B. Single-Point sVOLE

Single-point sVOLE is a variant of sVOLE where the vector of authenticated values contains exactly one nonzero entry. We present the associated functionality $\mathcal{F}^{p,r}_{\text{spsVOLE}}$ in Figure 6, where the vector length $n = 2^h$ is assumed to be a power of two for simplicity. In Figure 7, we present a protocol $\Pi^{p,r}_{\text{spsVOLE}}$ that UC-realizes $\mathcal{F}^{p,r}_{\text{spsVOLE}}$ in the $(\mathcal{F}^{p,r}_{\text{sVOLE}}, \mathcal{F}_{\text{OT}}, \mathcal{F}_{\text{EQ}})$-hybrid model, where $\mathcal{F}_{\text{OT}}$ is the standard OT functionality and $\mathcal{F}_{\text{EQ}}$ corresponds to a weak equality test that reveals $\mathsf{P_A}$'s input to $\mathsf{P_B}$. (See Appendix A for formal definitions of both functionalities.) Conceptually, the protocol can be divided into two steps: (1) the parties run a semi-honest protocol for generating a vector of authenticated values $[\boldsymbol{u}]$ having a single nonzero entry; then (2) a consistency check is performed to detect malicious behavior. We explain both steps in what follows.

$\mathsf{P_A}$ begins by choosing a uniform $\beta \in \mathbb{F}_p^*$ and a uniform index α. Letting $\boldsymbol{u} \in \mathbb{F}_p^n$ be the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] = \beta$, the goal is for the parties to generate $[\boldsymbol{u}]$.

Functionality $\mathcal{F}^{p,r}_{\text{spsVOLE}}$

Initialize: Upon receiving init from $\mathsf{P_A}$ and $\mathsf{P_B}$, sample $\Delta \leftarrow \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest and receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ, send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands.

Extend: Upon receiving (sp-extend, n), where $n = 2^h$ for some $h \in \mathbb{N}$, from $\mathsf{P_A}$ and $\mathsf{P_B}$, do:

1) If $\mathsf{P_B}$ is honest, sample $\boldsymbol{v} \leftarrow \mathbb{F}_{p^r}^n$. Otherwise, receive $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ from the adversary.
2) If $\mathsf{P_A}$ is honest, then sample uniform $\boldsymbol{u} \in \mathbb{F}_p^n$ with exactly one nonzero entry, and compute $\boldsymbol{w} := \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}_{p^r}^n$. Otherwise, receive $\boldsymbol{u} \in \mathbb{F}_p^n$ (with at most one nonzero entry) and $\boldsymbol{w} \in \mathbb{F}_{p^r}^n$ from the adversary, and recompute $\boldsymbol{v} := \boldsymbol{w} - \Delta \cdot \boldsymbol{u} \in \mathbb{F}_{p^r}^n$.
3) If $\mathsf{P_B}$ is corrupted, receive a set $I \subseteq [0, n)$ from the adversary. Let $\alpha \in [0, n)$ be the index of the nonzero entry of $\boldsymbol{u}$. If $\alpha \in I$, send success to $\mathsf{P_B}$ and continue. Otherwise, send abort to both parties and abort.
4) Send $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathsf{P_A}$ and $\boldsymbol{v}$ to $\mathsf{P_B}$.

Global-key query: If $\mathsf{P_A}$ is corrupted, receive (guess, Δ') from the adversary with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send success to $\mathsf{P_A}$ and ignore any subsequent global-key query. Otherwise, send abort to both parties and abort.

Fig. 6: **Functionality for single-point sVOLE.**

That is, they want $\mathsf{P_A}$ to hold $\boldsymbol{w} \in \mathbb{F}_{p^r}^n$ and $\mathsf{P_B}$ to hold $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$. To do so, the parties begin by generating the authenticated value $[\beta]$; this is easy to do using a call to $\mathcal{F}^{p,r}_{\text{sVOLE}}$. Next, they use a subroutine [17], [18], [16] based on the GGM construction [35] to enable $\mathsf{P_B}$ to generate $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ while allowing $\mathsf{P_A}$ to learn all the components of that vector except for $\boldsymbol{v}[\alpha]$. This is done in the following way. Let $G : \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$ and $G' : \{0,1\}^\kappa \to \mathbb{F}_{p^r}^2$ be pseudorandom generators (PRGs). $\mathsf{P_B}$ chooses uniform $s \in \{0,1\}^\kappa$ and computes all nodes in a GGM tree of depth h with s at the root: That is, letting s^i_j denote the value at the jth node on the ith level of the tree, $\mathsf{P_B}$ defines $s^0_0 := s$ and then for $i \in [1, h)$ and $j \in [0, 2^{i-1})$ computes $\left(s^i_{2j}, s^i_{2j+1}\right) := G(s^{i-1}_j)$; finally, $\mathsf{P_B}$ computes a vector $\boldsymbol{v}$ at the leaves as $(\boldsymbol{v}[2j], \boldsymbol{v}[2j+1]) := G'(s^{h-1}_j)$ for $j \in [0, 2^{h-1})$. Next, $\mathsf{P_B}$ lets K^i_0 (resp., K^i_1) be the XOR of the values at the even (resp., odd) nodes on the ith level. (When $i = h$ we replace XOR with addition in $\mathbb{F}_{p^r}$.) We write

$$\left(\{v_j\}_{j\in[0,n)},\ \{(K^i_0, K^i_1)\}_{i\in[h]}\right) := \mathsf{GGM}(1^n, s)$$

to denote this computation done by $\mathsf{P_B}$. It is easily verified that if $\mathsf{P_A}$ is given $\{K^i_{\bar{\alpha}_i}\}_{i\in[h]}$ (where $\bar{\alpha}_i$ is the complement of the ith bit of α), then $\mathsf{P_A}$ can compute $\{\boldsymbol{v}[j]\}_{j\neq\alpha}$, while $\boldsymbol{v}[\alpha]$ remains computationally indistinguishable from uniform given $\mathsf{P_A}$'s view. We denote the resulting computation of $\mathsf{P_A}$ by $\{v_j\}_{j\neq\alpha} := \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i\in[h]})$. ($\mathsf{P_A}$ can obtain $\{K^i_{\bar{\alpha}_i}\}_{i\in[h]}$ using h OT invocations.)

Following the above, $\mathsf{P_A}$ sets $\boldsymbol{w}[i] := \boldsymbol{v}[i]$ for $i \neq \alpha$. Note that $\boldsymbol{w}[i] = \boldsymbol{v}[i] + \Delta \cdot \boldsymbol{u}[i]$ for $i \neq \alpha$ (since $\boldsymbol{u}[i] = 0$ for $i \neq \alpha$), so all that remains is for $\mathsf{P_A}$ to obtain the missing value $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$ (without revealing α, β to $\mathsf{P_B}$). Recall the parties already hold $[\beta]$, meaning that $\mathsf{P_A}$ holds

Protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$

Initialize: This procedure is executed only once.

- $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns Δ to $\mathsf{P_B}$.

Extend: This procedure can be run multiple times. On input $n = 2^h$, the parties do:

1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send $(\mathsf{extend}, 1)$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(a, c) \in \mathbb{F}_p \times \mathbb{F}_{p^r}$ to $\mathsf{P_A}$ and $b \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$ such that $c = b + \Delta \cdot a$. Then, $\mathsf{P_A}$ samples $\beta \leftarrow \mathbb{F}^*_p$, sets $\delta := c$, and sends $a' := \beta - a \in \mathbb{F}_p$ to $\mathsf{P_B}$, who computes $\gamma := b - \Delta \cdot a'$. Note that $\delta = \gamma + \Delta \cdot \beta \in \mathbb{F}_{p^r}$, so the parties now hold $[\beta]$. $\mathsf{P_A}$ samples $\alpha \leftarrow [0, n)$ and defines $\boldsymbol{u} \in \mathbb{F}^n_p$ as the vector that is 0 everywhere except $\boldsymbol{u}[\alpha] = \beta$.
2) $\mathsf{P_B}$ samples $s \leftarrow \{0,1\}^\kappa$, runs $\mathsf{GGM}(1^n, s)$ to obtain $\left(\{v_j\}_{j\in[0,n)}, \{(K^i_0, K^i_1)\}_{i\in[h]}\right)$, and sets $\boldsymbol{v}[j] := v_j$ for $j \in [0, n)$. $\mathsf{P_A}$ lets $\bar{\alpha}_i$ be the complement of the ith bit of the binary representation of α. For $i \in [h]$, $\mathsf{P_A}$ sends $\bar{\alpha}_i \in \{0, 1\}$ to $\mathcal{F}_{\mathsf{OT}}$ and $\mathsf{P_B}$ sends (K^i_0, K^i_1) to $\mathcal{F}_{\mathsf{OT}}$, which returns $K^i_{\bar{\alpha}_i}$ to $\mathsf{P_A}$. Then $\mathsf{P_A}$ runs $\{v_j\}_{j\neq\alpha} := \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i\in[h]})$.
3) $\mathsf{P_B}$ sends $d := \gamma - \sum_{i\in[0,n)} \boldsymbol{v}[i] \in \mathbb{F}_{p^r}$ to $\mathsf{P_A}$. Then, $\mathsf{P_A}$ defines $\boldsymbol{w} \in \mathbb{F}^n_{p^r}$ as the vector with $\boldsymbol{w}[i] := v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] := \delta - \left(d + \sum_{i\neq\alpha} \boldsymbol{w}[i]\right)$. Note that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$.

Consistency check:

4) Both parties send (extend, r) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}^r_p \times \mathbb{F}^r_{p^r}$ to $\mathsf{P_A}$ and $\boldsymbol{y}^* \in \mathbb{F}^r_{p^r}$ to $\mathsf{P_B}$ such that $\boldsymbol{z} = \boldsymbol{y}^* + \Delta \cdot \boldsymbol{x}$.
5) $\mathsf{P_A}$ samples $\chi_i \leftarrow \mathbb{F}_{p^r}$ for $i \in [0, n)$, and writes $\chi_\alpha = \sum^{r-1}_{i=0} \chi_{\alpha,i} \cdot \mathsf{X}^i$. Let $\boldsymbol{\chi}_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}^r_p$. $\mathsf{P_A}$ then computes $\boldsymbol{x}^* := \beta \cdot \boldsymbol{\chi}_\alpha - \boldsymbol{x} \in \mathbb{F}^r_p$ and sends $\left(\{\chi_i\}_{i\in[0,n)}, \boldsymbol{x}^*\right)$ to $\mathsf{P_B}$, who computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^*$.
6) $\mathsf{P_A}$ computes $Z := \sum^{r-1}_{i=0} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_{\mathsf{A}} := \sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{w}[i] - Z \in \mathbb{F}_{p^r}$, while $\mathsf{P_B}$ computes $Y := \sum^{r-1}_{i=0} \boldsymbol{y}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_{\mathsf{B}} := \sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{v}[i] - Y \in \mathbb{F}_{p^r}$. Then $\mathsf{P_A}$ sends V_{A} to $\mathcal{F}_{\mathsf{EQ}}$, and $\mathsf{P_B}$ sends V_{B} to $\mathcal{F}_{\mathsf{EQ}}$. If either party receives false or abort from $\mathcal{F}_{\mathsf{EQ}}$, it aborts.
7) $\mathsf{P_A}$ outputs $(\boldsymbol{u}, \boldsymbol{w})$ and $\mathsf{P_B}$ outputs $\boldsymbol{v}$.

Fig. 7: **Single-point sVOLE protocol.**

$\mathsf{M}[\beta]$ and $\mathsf{P_B}$ holds $\mathsf{K}[\beta]$ with $\mathsf{M}[\beta] = \mathsf{K}[\beta] + \Delta \cdot \beta$. So if $\mathsf{P_B}$ sends $\mathsf{K}[\beta] - \sum_i \boldsymbol{v}[i]$, then $\mathsf{P_A}$ can compute the missing value: $\boldsymbol{w}[\alpha] = \mathsf{M}[\beta] - (\mathsf{K}[\beta] - \sum_i \boldsymbol{v}[i]) - \sum_{i\neq\alpha} \boldsymbol{v}[i] = \mathsf{M}[\beta] - \mathsf{K}[\beta] + \boldsymbol{v}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$. This completes the "semi-honest" portion of the protocol.

To verify correct behavior, we generalize the approach of Yang et al. [60] that applies only to the case $p = 2$. We want to verify that $\boldsymbol{w}[i] = \boldsymbol{v}[i]$ for $i \neq \alpha$, and $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$. Intuitively, the parties do this by having $\mathsf{P_A}$ choose uniform $\chi_0, \ldots, \chi_{n-1} \in \mathbb{F}_{p^r}$ and then checking that

$$\sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{w}[i] = \sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{v}[i] + \Delta \cdot \beta \cdot \chi_\alpha.$$

Of course, this must be done without revealing α, β to $\mathsf{P_B}$. To do so, $\mathsf{P_A}$ and $\mathsf{P_B}$ use $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ to compute $Z, Y \in \mathbb{F}_{p^r}$, respectively, such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. (We discuss below how this is done.) They then use $\mathcal{F}_{\mathsf{EQ}}$ to check if $V_{\mathsf{A}} = \sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{w}[i] - Z$ is equal to $V_{\mathsf{B}} = \sum^{n-1}_{i=0} \chi_i \cdot \boldsymbol{v}[i] - Y$.

To complete the description, we show how the parties can generate Z, Y (held by $\mathsf{P_A}$, $\mathsf{P_B}$, respectively) such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. (This is like an authenticated value $[\beta \cdot \chi_\alpha]$, but note that $\beta \cdot \chi_\alpha$ lies in $\mathbb{F}_{p^r}$ rather than $\mathbb{F}_p$.) $\mathsf{P_A}$ views $\chi_\alpha \in \mathbb{F}_{p^r}$ as $\boldsymbol{\chi}_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}^r_p$ (i.e., $\chi_\alpha = \sum_{i\in[0,r)} \chi_{\alpha,i} \cdot \mathsf{X}^i$, where $\{\mathsf{X}^i\}_{i\in[0,r)}$ form a basis for $\mathbb{F}_{p^r}$ over $\mathbb{F}_p$), and then the parties use $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ to generate the vector of authenticated values $[\beta \cdot \boldsymbol{\chi}_\alpha]$. This means $\mathsf{P_A}$ holds $\boldsymbol{z}$ and $\mathsf{P_B}$ holds $\boldsymbol{y}$ such that $\boldsymbol{z} = \boldsymbol{y} + \Delta \cdot \beta \cdot \boldsymbol{\chi}_\alpha$. Let $Z = \sum_{i\in[0,r)} \boldsymbol{z}[i] \cdot \mathsf{X}^i$ and $Y = \sum_{i\in[0,r)} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. We have that

$$\begin{aligned} Z &= \sum^{r-1}_{i=0} \boldsymbol{z}[i] \cdot \mathsf{X}^i = \sum^{r-1}_{i=0} (\boldsymbol{y}[i] + \Delta \cdot \beta \cdot \boldsymbol{\chi}_\alpha[i]) \cdot \mathsf{X}^i \\ &= \sum^{r-1}_{i=0} \boldsymbol{y}[i] \cdot \mathsf{X}^i + \Delta \cdot \beta \cdot \sum^{r-1}_{i=0} \boldsymbol{\chi}_\alpha[i] \cdot \mathsf{X}^i = Y + \Delta \cdot \beta \cdot \chi_\alpha, \end{aligned}$$

as desired.

We remark that this check allows a malicious $\mathsf{P_A}$ to guess Δ, and allows a malicious $\mathsf{P_B}$ to guess a subset in which the index α lies. (This will become evident in the proof of security.) Such guesses are incorporated into the ideal functionality $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$.

Theorem 3. *If G and G' are pseudorandom generators, then $\Pi^{p,r}_{\mathsf{spsVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model. In particular, no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most $1/p^r + \mathsf{negl}(\kappa)$.*

The proof of Theorem 3 is given in Appendix D.

Optimizations. We discuss various optimizations of the protocol shown in Figure 7:

1) For large p (i.e., $\log p \geq \rho$), the parties can use the output of $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ directly as $[\beta]$ in step 1 of protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$, since $\beta \neq 0$ with overwhelming probability.
2) In the consistency check, $\mathsf{P_A}$ can send uniform seed $\in \{0,1\}^\kappa$ to $\mathsf{P_B}$, who then derives the $\{\chi_i\}$ from seed using a hash function modeled as a random oracle.
3) When t extend executions are needed, we can batch the consistency checks using the ideas of Yang et al. [60] to reduce the total number of sVOLE correlations needed from $t \cdot (1 + r)$ to $t + r$. The approach is as follows:
   a) After t executions of the semi-honest portion of the extend phase, the parties hold $\{(\boldsymbol{u}_j, \boldsymbol{w}_j)\}^t_{j=1}$ and $\{\boldsymbol{v}_j\}^t_{j=1}$, respectively, where for all $j \in [t]$ we have $\boldsymbol{w}_j = \boldsymbol{v}_j + \Delta \cdot \boldsymbol{u}_j$ with $\boldsymbol{u}_j$ a vector that is 0 everywhere except $\boldsymbol{u}_j[\alpha_j] = \beta_j$. Then $\mathsf{P_A}$ and $\mathsf{P_B}$ send (extend, r) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z})$ to $\mathsf{P_A}$ and $\boldsymbol{y}^*$ to $\mathsf{P_B}$.
   b) For $j \in [t]$, $\mathsf{P_A}$ samples $\chi_{i,j} \leftarrow \mathbb{F}_{p^r}$ for $i \in [0, n)$, and views $\chi_{\alpha_j,j}$ as the vector $\boldsymbol{\chi}_{\alpha_j,j} \in \mathbb{F}^r_p$. It then computes $\boldsymbol{x}^* := \sum_{j\in[t]} \beta_j \cdot \boldsymbol{\chi}_{\alpha_j,j} - \boldsymbol{x}$ and sends $\{\chi_{i,j}\}_{i\in[0,n),j\in[t]}$ and $\boldsymbol{x}^*$ to $\mathsf{P_B}$, who computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^* \in \mathbb{F}^r_{p^r}$.
   c) $\mathsf{P_A}$ computes $V_{\mathsf{A}} := \sum^{n-1}_{i=0} \sum^t_{j=1} \chi_{i,j} \cdot \boldsymbol{w}_j[i] - \sum^{r-1}_{i=0} \boldsymbol{z}[i] \cdot \mathsf{X}^i$; $\mathsf{P_B}$ computes $V_{\mathsf{B}} := \sum^{n-1}_{i=0} \sum^t_{j=1} \chi_{i,j} \cdot \boldsymbol{v}_j[i] - \sum^{r-1}_{i=0} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. Then both parties check whether $V_{\mathsf{A}} = V_{\mathsf{B}}$ by calling $\mathcal{F}_{\mathsf{EQ}}$.

C. sVOLE Extension

We show here a protocol that can be viewed as a means of performing "sVOLE extension." That is, our protocol allows two parties to efficiently extend a small number of sVOLE correlations (created in a setup phase) to an arbitrary polynomial number of sVOLE correlations. The protocol relies on spsVOLE as a subroutine, as well as a variant of the LPN assumption that has been used in prior work [41], [14], [60].

Protocol overview. The parties use the base-sVOLE protocol to generate a length-k vector of authenticated values $[\boldsymbol{u}]$. They also use spsVOLE to generate t vectors of authenticated values, each of length n/t and having a single nonzero entry; they let $[\boldsymbol{e}]$ be the concatenation of those vectors. The parties then use a public matrix $\mathbf{A}$ to define the length-n vector of authenticated values $[\boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e}]$; by the LPN assumption, the corresponding values (which $\mathsf{P_A}$ knows) will appear pseudorandom to $\mathsf{P_B}$. This provides a way to extend k random sVOLE correlations to n pseudorandom sVOLE correlations once. As in prior work [60], however, we can generate $\ell = n - k$ correlations as many times as desired by simply using this idea to generate n sVOLE correlations and reserving the first k of those correlations for the next iteration of the extend phase.

LPN assumption. Let $\mathcal{D}_{n,t}$ denote the distribution over an error vector $\boldsymbol{e} \in \mathbb{F}_p^n$ in which $\boldsymbol{e}$ is divided into t blocks (each of length n/t), and each block of contains exactly one uniform nonzero entry at a uniform location within that block.

Definition 1 (LPN with static leakage [14])**.** *Let $\mathcal{G}$ be a polynomial-time algorithm that on input $1^k, 1^n, p$ outputs $\mathbf{A} \in \mathbb{F}_p^{k \times n}$. Let parameters k, n, t be implicit functions of security parameter κ. We say that the $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ assumption holds if for all PPT algorithms $\mathcal{A}$ we have*

$$\left| \Pr[\mathsf{LPN\text{-}Succ}^{\mathcal{G}}_{\mathcal{A}}(\kappa) = 1] - 1/2 \right| \leq \mathsf{negl}(\kappa),$$

where the experiment $\mathsf{LPN\text{-}Succ}^{\mathcal{G}}_{\mathcal{A}}(\kappa)$ is defined as follows:

1) *Sample $\mathbf{A} \leftarrow \mathcal{G}(1^k, 1^n, p)$, $\boldsymbol{u} \leftarrow \mathbb{F}_p^k$, and $\boldsymbol{e} \leftarrow \mathcal{D}_{n,t}$. Let $\alpha_1, \ldots, \alpha_t$ be the indices of the nonzero entries in $\boldsymbol{e}$ (each of which is located in a disjoint block of length n/t).*
2) *$\mathcal{A}$ outputs t subsets $I_1, \ldots, I_t \subseteq [0, n)$. If $\alpha_i \in I_i$ for all $i \in [t]$, then send $\mathsf{success}$ to $\mathcal{A}$; otherwise, abort the experiment and define $b' := 0$.*
3) *Pick $b \leftarrow \{0, 1\}$. If $b = 0$, let $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e}$; otherwise, sample $\boldsymbol{x} \leftarrow \mathbb{F}_p^n$. Send $\boldsymbol{x}$ to $\mathcal{A}$, who then outputs a bit b' (if the experiment did not abort).*
4) *The experiment outputs 1 iff $b' = b$.*

Protocol description. In Figure 8, we present our sVOLE extension protocol in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}^{p,r}_{\mathsf{spsVOLE}})$-hybrid model. For simplicity, we assume a public matrix $\mathbf{A} \in \mathbb{F}_p^{k \times n}$, output by an efficient algorithm $\mathcal{G}(1^k, 1^n, p)$, that is fixed at the outset of the protocol. (It is also possible to have $\mathsf{P_A}$ generate $\mathbf{A}$ and then send it to $\mathsf{P_B}$.) We assume that $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ share the same initialization (i.e., use the same global key Δ). This holds, in particular, when we use protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$ from the previous section to UC-realize $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$.

Protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$

Parameters: Fix n, k, t, and define $\ell = n - k$ and $m = n/t$. Let $\mathbf{A} \in \mathbb{F}_p^{k \times n}$ be a matrix output by $\mathcal{G}(1^k, 1^n, p)$.

Initialize: This procedure is executed only once.

1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns Δ to $\mathsf{P_B}$.
2) $\mathsf{P_A}$ and $\mathsf{P_B}$ send (extend, k) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathsf{P_A}$ and $\boldsymbol{v}$ to $\mathsf{P_B}$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}_{p^r}^k$.

Extend: This procedure can be executed multiple times.

3) For $i \in [t]$, $\mathsf{P_A}$ and $\mathsf{P_B}$ send $(\mathsf{sp\text{-}extend}, m)$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$, which returns $(\boldsymbol{e}_i, \boldsymbol{c}_i)$ to $\mathsf{P_A}$ and $\boldsymbol{b}_i$ to $\mathsf{P_B}$ such that $\boldsymbol{c}_i = \boldsymbol{b}_i + \Delta \cdot \boldsymbol{e}_i \in \mathbb{F}_{p^r}^m$ and $\boldsymbol{e}_i \in \mathbb{F}_p^m$ has exactly one nonzero entry. If either party receives abort from $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ in any of these spsVOLE executions, it aborts.
4) $\mathsf{P_A}$ defines $\boldsymbol{e} = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^n$ and $\boldsymbol{c} = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^n$. Then $\mathsf{P_A}$ computes $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e} \in \mathbb{F}_p^n$ and $\boldsymbol{z} := \boldsymbol{w} \cdot \mathbf{A} + \boldsymbol{c} \in \mathbb{F}_{p^r}^n$. $\mathsf{P_B}$ defines $\boldsymbol{b} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^n$ and computes $\boldsymbol{y} := \boldsymbol{v} \cdot \mathbf{A} + \boldsymbol{b} \in \mathbb{F}_{p^r}^n$.
5) $\mathsf{P_A}$ updates $\boldsymbol{u}, \boldsymbol{w}$ by setting $\boldsymbol{u} := \boldsymbol{x}[0:k) \in \mathbb{F}_p^k$ and $\boldsymbol{w} := \boldsymbol{z}[0:k) \in \mathbb{F}_{p^r}^k$, and outputs $(\boldsymbol{s}, \mathsf{M}[\boldsymbol{s}]) := (\boldsymbol{x}[k:n), \boldsymbol{z}[k:n)) \in \mathbb{F}_p^\ell \times \mathbb{F}_{p^r}^\ell$. $\mathsf{P_B}$ updates $\boldsymbol{v}$ by setting $\boldsymbol{v} := \boldsymbol{y}[0:k) \in \mathbb{F}_{p^r}^k$, and outputs $\mathsf{K}[\boldsymbol{s}] := \boldsymbol{y}[k:n) \in \mathbb{F}_{p^r}^\ell$.

Fig. 8: **The sVOLE extension protocol.**

Theorem 4. *If the $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ assumption holds, then $\Pi^{p,r}_{\mathsf{sVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}^{p,r}_{\mathsf{spsVOLE}})$-hybrid model.*

A proof of the above can be found in Appendix E, where we also describe further optimizations for protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$.

V. Performance Evaluation

In this section, we report on the performance of our sVOLE protocol and our overall ZK protocol for both boolean and arithmetic circuits. All our protocols were implemented in the EMP toolkit [57], and we will release an open-source version of our code. In all our experiments, we use two Amazon EC2 instances of type `m5.4xlarge` with 16 vCPUs and 64 GB of RAM, using 5 threads. We artificially limit the network bandwidth as indicated in each experiment. All implementations achieve the statistical security parameter $\rho \geq 40$ and computational security parameter $\kappa = 128$.

A. (Subfield) Vector Oblivious Linear Evaluation

We focus here on the performance of protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ over large fields; specifically, we fix the Mersenne prime $p = 2^{61} - 1$ and set $r = 1$. (sVOLE is equivalent to VOLE in this case.)

Parameter selection. As suggested in prior work [13], [53], [60], we choose the public LPN matrix $\mathbf{A}$ as a generator of a 10-local linear code, which means that each column of $\mathbf{A}$ contains exactly 10 (uniform) nonzero entries. This is advantageous since it means that computing each entry of $\boldsymbol{u} \cdot \mathbf{A}$ involves reading only 10 positions of $\boldsymbol{u} \in \mathbb{F}_p^k$. To ensure that reading those positions can be done quickly, we set k so that $\boldsymbol{u}$ fits in the L1 CPU cache (i.e., the size of $\boldsymbol{u}$ is less than 8 MB). With k fixed, for any choice of $n > k$ we can take the

One-time setup			Extend execution		
k_0	n_0	t_0	k	n	t
19,870	642,048	2,508	589,760	10,805,248	1,319

TABLE II: **LPN parameters used in our VOLE protocol.**

	20 Mbps	50 Mbps	100 Mbps	500 Mbps	1 Gbps
Init. (*ms*)	1343	640	478	451	438
Extend (*ns*/VOLE)	101	87	85	85	85

TABLE III: **Efficiency of our VOLE protocol as a function of network bandwidth.** The communication per VOLE correlation is 0.42 bits; the overall communication of the one-time setup is 1.1 MB.

smallest t for which all known attacks on the LPN problem require at least 2^{128} operations [13], [14]. When we apply the optimizations described at the end of the previous section to our protocol, we see that using LPN parameters (n, k, t) means that each invocation of the extend procedure results in $n-k-t-1$ usable VOLE correlations. We perform exhaustive search to find the smallest n so that $n-k-t-1 \geq 10^7$. For the parameters of the setup phase, we follow the same step as above, except that we will ensure that $n_0 - k_0 - t_0 - 1 \geq k$. This results in the LPN parameters shown in Table II.

Performance. We evaluate the efficiency of protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ in Table III. The extend procedure requires very little communication (less than half a bit per usable VOLE correlation), and its execution time is largely unaffected by the network bandwidth above 100 Mbps. The one-time initialization only communicates 1.1 MB and takes roughly 478 milliseconds under a 100 Mbps network.

In Table IV, we compare our VOLE protocol with the best known protocols that have been implemented [53], [27]. Since our protocol needs an one-time setup, that can be amortized over multiple executions, we report our performance both without one-time setup (in case multiple extensions are executed), and the one with one-time setup (in case only one extension is executed). We fix the network bandwidth to 500 Mbps to match the experiments of Castro et al. [27]. Our protocol outperforms prior work even though prior work is secure only against *semi-honest* adversaries, whereas our protocol is secure in the malicious setting. Note in particular that the communication complexity of our protocol is orders of magnitude lower than prior work. Boyle et al. [14] also proposed a maliciously secure sVOLE protocol but only implemented their protocol for the special case $p = 2, r = 128$. Based on their implementation in that case, we estimate that

	[53]	[27]	Ours (w/o setup)	Ours (w/ setup)
Communication (bits)	960	160	0.42	1.32
Execution time (*ns*)	2000	400	85	130

TABLE IV: **Our VOLE protocol vs. prior protocols.** We fix the network bandwidth to 500 Mbps and report the marginal cost per VOLE correlation. Running time for the protocol of Schoppmann et al. [53] is the time for communication alone; numbers for the protocol of Castro et al. [27] are taken from their paper and are based on the same network and CPU configuration but using 8 threads.

for our choice of p their protocol would communicate roughly 0.14 bits per sVOLE; however their computation is much heavier than ours and would take time at least 900 *ns* per VOLE correlation. Therefore, we believe that our protocol is still more efficient for most network bandwidth settings.

B. Zero-Knowledge Proofs

We report on the performance of our ZK protocol for boolean and arithmetic circuits. In both cases, we use pipelining [43] to streamline the protocol execution. This significantly reduces the memory usage (from linear in the circuit size to linear in the memory needed to evaluate the circuit non-cryptographically) and allows us to scale to very large circuits.

To further reduce the memory usage for large circuits, we changed the protocol so that rather than checking the correctness of all C multiplication gates at the end, we check blocks of $C' < C$ gates at a time. This increases the round complexity to $\mathcal{O}(C/C')$ but reduces the memory usage to $\mathcal{O}(C')$.

1) Zero-Knowledge Proofs for Boolean Circuits: In the boolean setting, we check correctness of multiplication gates as in Figure 3. Theorem 1 shows that to achieve ρ-bit statistical security we need $\binom{C'B+c}{B} > 2^\rho$. Setting $c = B$, we have

$$\binom{C'B+B}{B} = \prod_{i=0}^{B-1} \left(\frac{B}{B-i} \cdot C' + 1 \right) \geq C'^B,$$

and so we need $C' \geq 2^{\rho/B}$. For the best efficiency, we set $B = 2$ when possible. For batched opening of authenticated values, we use the second approach described in Section II-A along with the Fiat-Shamir heuristic to make it non-interactive. We instantiate $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ using Ferret [60] with $p = 2$ and $r = 128$.

Performance. The execution of our protocol can be split into two stages: input processing, whose cost is proportional to the witness length, and circuit processing, whose cost is proportional to the number of AND gates. Therefore, we measure the scalability of our ZK protocol by increasing either the witness length or the circuit size while artificially keeping the other value fixed. The experimental results in Figure 9 show that the execution time is indeed linear in both the witness length and the circuit size, with very small marginal cost for each bit of the witness or each AND gate. For example, under a 50 Mbps network, the marginal time of our protocol is 3.35 μs per bit of the witness and 0.5 μs per AND gate of the circuit.

The ZKGC approach [47], [42] is the only previous approach for efficient ZK proofs that scales to large circuits while using less than 10 GB of memory. The communication complexity of our protocol is roughly 15× lower than the ZKGC approach. For this reason, our protocol is particularly well-suited for settings involving a low-bandwidth network.

Example 1: Merkle trees. As a representative example highlighting the efficiency and scalability of our protocol, we consider proving knowledge of the $n = 2^d$ leaves in a complete Merkle tree of depth d using SHA-256 as the

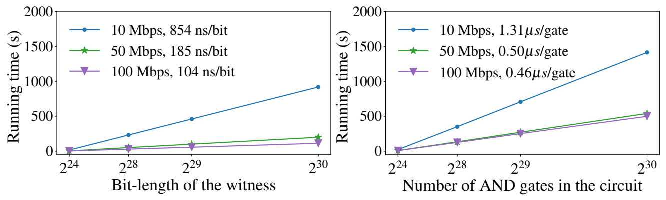

Fig. 9: **Scalability of** Wolverine **for boolean circuits.**

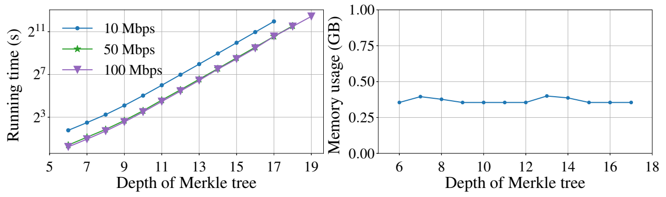

Fig. 10: **Running time and memory usage of** Wolverine **when proving knowledge of all leaves in a Merkle tree of a given depth.**

internal hash function, where the root digest is known to both parties. In Figure 10, we report on the running time and overall memory usage of our protocol for d ranging from 6 to 19 (totaling 63–524,287 calls to SHA-256). Since the boolean circuit for SHA-256 has 22,573 AND gates, the largest circuit in these experiments contains more than 11 billion gates.

The overall memory consumption of our protocol is about 400 MB; this is dominated by the initial generation of 10^7 sVOLE correlations during the offline phase of the execution. During the online phase, the Merkle-tree computation is implemented in a post-order, depth-first fashion so that the additional memory usage at any point corresponds only to authenticated values for $\mathcal{O}(d)$ tree nodes (at most 150 KB). Since this is dominated by the memory usage during the online phase, the memory usage plotted in Figure 10 is nearly constant even as d increases.

In Table I, we compare the performance of our protocol to that of the state-of-the-art protocols for the same problem, in a 200 Mbps network. (There, we report on the performance of our protocol using only one thread.) We benchmarked all prior work except for Ligero [1], for which we obtained performance estimates from the authors. Spartan [54] uses the R1CS representation, so we conservatively assume that each SHA-256 hash requires 22,573 constraints. Virgo [63] does not support free-XOR, and thus for each SHA-256 hash it uses roughly 2^{18} gates. (For this reason, the running time of Virgo for the Merkle-tree example is close to its running time for the matrix-multiplication example.) Compared to the ZKGC approach, Wolverine achieves better running time and about $15\times$ lower communication, which means that it will be up to $15\times$ faster when running in a low-bandwidth network. Compared to other protocols, we achieve at least a $5\times$ improvement in execution time while using less memory.

Example 2: Proving existence of a bug in programs. We also apply our system to prove the existence of a bug in one out of a set of n program snippets in zero knowledge (in particular, without revealing which snippet contains the bug). This

	10 Mbps			100 Mbps		
Number of snippets	4	50	200	4	50	200
Stacked garbling (s)	22	22.1	22.2	2.3	2.5	3.18
Our protocol (s)	0.42	5.2	20.8	0.15	1.8	7.2

TABLE V: Wolverine **vs. ZKGC with stacked garbling for proving the existence of a bug in one of multiple code snippets.**

	DECO [62]		Blind CA [56]	
Protocol	DECO	Wolverine	Blind CA	Wolverine
Execution time	12.6 s	0.28 s	71 s	3.3 s
Communication	1.7 KB	184 KB	85.1 MB	2.8 MB

TABLE VI: **Using our protocol** Wolverine **in ZK-enabled applications.** All benchmarks are based on a 10 Mbps network and reflect the ZK component only.

problem was recently studied by Heath and Kolesnikov [42], who showed how to adapt the ZKGC approach using a technique called *stacked garbling* so as to obtain communication proportional to the size ℓ of the largest program snippet, rather than the total size $\mathcal{O}(n \cdot \ell)$ of all programs snippets.

We performed experiments using the same programs as in the work of Heath and Kolesnikov. These result in boolean circuits whose sizes range from 70,869–90,772 AND gates and whose largest input length is 112 bits. We show the results in Table V. Wolverine does not use the stacked garbling optimization,[2] and so has communication complexity $\mathcal{O}(n \cdot \ell)$. Nevertheless, for moderate values of n, Wolverine is still noticeably faster than ZKGC with stacked garbling. The effect is more pronounced in lower-bandwidth networks.

Example 3: Accelerating ZK-enabled applications. Here we discuss the use of Wolverine in two recent applications that rely on ZK proofs. Both applications require interaction anyway, and so there is no real disadvantage to using an interactive ZK proof in these cases. We describe the applications below, and present the relevant benchmarking results in Table VI.

DECO [62] allows third-party proofs of data provenance for TLS connections, i.e., it allows a client to prove that certain data originated at a particular website. (We confirmed with the authors that interactive ZK proofs can also be used in their system.) One example considered by DECO is where a customer proves the existence of price discrimination by proving in zero knowledge that it was sent a price exceeding a certain threshold. Proving this statement involves a boolean circuit containing roughly 163,000 AND gates. In the original paper [62], a ZK proof for this statement was implemented using libSNARK; this resulted in a short proof but required high computational overhead. When running over a 10 Mbps network, Wolverine is able to reduce the execution time of the ZK-proof component by $45\times$, resulting in a $9\times$ end-to-end improvement in the overall DECO protocol.

A blind Certificate Authority (CA) is able to issue a valid certificate binding a party with an associated public key, without learning the party's identity. A recent proposal of a blind CA [56] required a ZK proof of a statement corresponding to

[2]We leave incorporating stacked garbling into Wolverine as a future work.

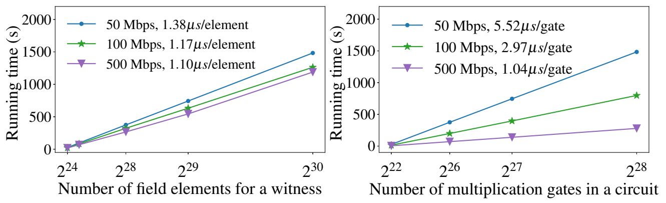

Fig. 11: **Performance of** Wolverine **for arithmetic circuits.**

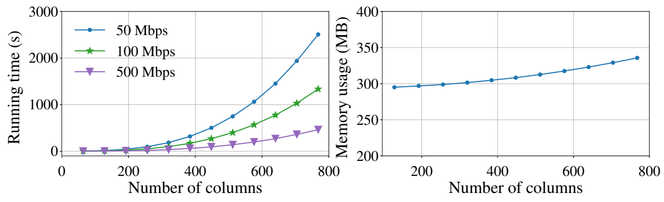

Fig. 12: **Using** Wolverine **for matrix multiplication.**

Our ZK protocol Wolverine (*ms*)				[4]
50 Mbps	100 Mbps	500 Mbps	10 Gbps	10 Gbps
74	63	55	55	1228

TABLE VII: **Running time of** Wolverine **vs. the protocol by Baum and Nof [4] for proving knowledge of an SIS solution.** The solution is assumed to be a binary vector.

Protocol	BLS [11]	Aurora [12]	ENS [30]	Ours
Communication	384 KB	233 KB	53 KB	32.8 KB

TABLE VIII: **Communication complexity of** Wolverine **vs. dedicated protocols for proving knowledge of an SIS solution.** The solution is assumed to be a vector over $\{-1, 0, 1\}$. Numbers for prior work are taken from Esgin et al. [30].

a boolean circuit with roughly 2.5 million AND gates. The existing implementation used a ZK proof based on the MPC-in-the-head approach; the proof took more than 70 seconds to execute over a 10 Mbps network. Plugging Wolverine into their protocol, we improve the communication complexity by $30\times$ and the execution time by $20\times$, compared to the original protocol [56] for CA-proof generation.

2) Zero-Knowledge Proofs for Arithmetic Circuits: We also evaluated Wolverine for arithmetic circuits over $\mathbb{F}_p$ with $p = 2^{61}-1$ using our VOLE implementation shown in Section V-A. In this setting we check correctness of multiplication gates using the first optimization described in Section III-B, and we use the first approach discussed in Section II-A for batched opening of authenticated values.

Performance. Similar to the boolean case, we study the performance of Wolverine as a function of the witness length and circuit size; the experimental results are reported in in Figure 11. As the communication complexity is inherently higher for the arithmetic case than the boolean setting (since each field element is 61 bits long), we benchmarked performance in higher-bandwidth networks. Wolverine can execute proofs at a rate of about 1 million multiplication gates per second in a 500 Mbps network, and roughly 200,000 multiplication gates per second in a 50 Mbps.

We are not aware of any memory-efficient ZK protocol that natively works with arithmetic circuits. While one could always convert an arithmetic circuit to a boolean circuit, this will generally impose significant overhead.

Example 1: Matrix multiplication. We apply our ZK protocol to prove knowledge of two $n \times n$ matrices whose product is a publicly known matrix. While the problem itself is meaningless, it has been used as a benchmark in prior work [10], [55], [58], [63]. We experimented with n ranging from 64–768 (with the witness ranging from 8,192 to over 1 million field elements), using a matrix-multiplication circuit corresponding to the naive $\mathcal{O}(n^3)$-time algorithm. The time and memory usage of Wolverine are shown in Figure 12. The memory usage of Wolverine grows slowly as n increases, and never exceeds 350 MB.

As shown in Table I, our protocol is $2\times$ faster than Spartan but $5\times$ slower than Virgo. Importantly, however, the prover memory of Wolverine is only 3% of that used by Virgo and 0.5% of that needed by Spartan.

Example 2: Solutions to lattice problems. Various prior works have explored ZK proofs for the Short Integer Solution (SIS) problem. Here, we have public $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and $\boldsymbol{t} \in \mathbb{Z}_q^n$, and the prover's goal is to convince the verifier that it knows a short $\boldsymbol{s}$ such that $\mathbf{A}\boldsymbol{s} = \boldsymbol{t} \bmod q$. We evaluate Wolverine based on different notions of shortness for $\boldsymbol{s}$ as explained next.

Baum and Nof [4] recently showed a ZK proof for SIS in the case where $\boldsymbol{s} \in \{0,1\}^m$ is a binary vector. We compare Wolverine to their protocol in Table VII. In our experiments, we use $q \approx 2^{61}$, $n = 1024$, and $m = 4096$ to align with the parameters used by Baum and Nof; those parameters are also sufficient for the somewhat homomorphic encryption scheme used for the SPDZ setup phase [19]. As shown in Table VII, our protocol is over $16\times$ more efficient than the protocol of Baum and Nof even when run over a much slower network.

In Table VIII, we compare Wolverine with other ZK proofs for SIS [11], [12], [30] that apply when $\boldsymbol{s} \in \{-1,0,1\}^m$. Here we fix $q \approx 2^{32}$, $n = 2048$, and $m = 1024$ to align with prior work. We see that Wolverine uses only 60% of the communication compared to the best prior work. (We are not able to compare the running time, since it was not reported by prior work.)

Acknowledgements

This material is based upon work supported in part by DARPA under Contract No. HR001120C0087. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Work of Kang Yang is supported by the National Natural Science Foundation of China (Grant No. 61932019). Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *ACM Conf.*

on Computer and Communications Security (CCS) 2017. ACM Press, 2017, pp. 2087–2104.

[2] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *IEEE Symp. Security and Privacy 2017*. IEEE, 2017, pp. 843–862.

[3] C. Baum, A. J. Malozemoff, M. Rosen, and P. Scholl, "Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions," Cryptology ePrint Archive, Report 2020/1410, 2020, https://eprint.iacr.org/2020/1410.

[4] C. Baum and A. Nof, "Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography," in *Intl. Conference on Theory and Practice of Public Key Cryptography 2020, Part I*, ser. LNCS. Springer, 2020, pp. 495–526.

[5] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Advances in Cryptology—Crypto 1991*, ser. LNCS. Springer, 1992, pp. 420–432.

[6] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable zero knowledge with no trusted setup," in *Advances in Cryptology—Crypto 2019, Part III*, ser. LNCS, vol. 11694. Springer, 2019, pp. 701–732.

[7] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Advances in Cryptology—Crypto 2013, Part II*, ser. LNCS, vol. 8043. Springer, 2013, pp. 90–108.

[8] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for R1CS," in *Advances in Cryptology—Eurocrypt 2019, Part I*, ser. LNCS, vol. 11476. Springer, 2019, pp. 103–128.

[9] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *USENIX Security Symposium 2014*. USENIX Association, 2014, pp. 781–796.

[10] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, "Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting," in *Advances in Cryptology—Eurocrypt 2016, Part II*, ser. LNCS, vol. 9666. Springer, 2016, pp. 327–357.

[11] J. Bootle, V. Lyubashevsky, and G. Seiler, "Algebraic techniques for short(er) exact lattice-based zero-knowledge proofs," in *Advances in Cryptology—Crypto 2019, Part I*, ser. LNCS, vol. 11692. Springer, 2019, pp. 176–202.

[12] C. Boschini, J. Camenisch, M. Ovsiankin, and N. Spooner, "Efficient post-quantum SNARKs for RSIS and RLWE and their applications to privacy," in *PQCrypto 2020*. Springer, Apr. 9–11 2020, pp. 247–267.

[13] E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai, "Compressing vector OLE," in *ACM Conf. on Computer and Communications Security (CCS) 2018*. ACM Press, 2018, pp. 896–912.

[14] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," in *ACM Conf. on Computer and Communications Security (CCS) 2019*. ACM Press, 2019, pp. 291–308.

[15] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent OT extension and more," in *Advances in Cryptology—Crypto 2019, Part III*, ser. LNCS, vol. 11694. Springer, 2019, pp. 489–518.

[16] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù, "Homomorphic secret sharing: Optimizations and applications," in *ACM Conf. on Computer and Communications Security (CCS) 2017*. ACM Press, 2017, pp. 2105–2122.

[17] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *Advances in Cryptology—Eurocrypt 2015, Part II*, ser. LNCS, vol. 9057. Springer, 2015, pp. 337–367.

[18] ——, "Function secret sharing: Improvements and extensions," in *ACM Conf. on Computer and Communications Security (CCS) 2016*. ACM Press, 2016, pp. 1292–1303.

[19] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS 2012*. Cambridge, MA, USA: Association for Computing Machinery, Jan. 8–10, 2012, pp. 309–325.

[20] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE Symp. Security and Privacy 2018*. IEEE, 2018, pp. 315–334.

[21] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent SNARKs from DARK compilers," in *Advances in Cryptology—Eurocrypt 2020, Part I*, ser. LNCS, vol. 12105. Springer, 2020, pp. 677–706.

[22] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2001, pp. 136–145.

[23] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha, "Post-quantum zero-knowledge and signatures from symmetric-key primitives," in *ACM Conf. on Computer and Communications Security (CCS) 2017*. ACM Press, 2017, pp. 1825–1842.

[24] A. Chiesa, D. Ojha, and N. Spooner, "Fractal: Post-quantum and transparent recursive proofs from holography," in *Advances in Cryptology—Eurocrypt 2020, Part I*, ser. LNCS, vol. 12105. Springer, 2020, pp. 769–793.

[25] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority," in *Advances in Cryptology—Crypto 2018, Part II*, ser. LNCS, vol. 10992. Springer, 2018, pp. 769–798.

[26] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Advances in Cryptology—Crypto 2012*, ser. LNCS, vol. 7417. Springer, 2012, pp. 643–662.

[27] L. de Castro, C. Juvekar, and V. Vaikuntanathan, "Fast vector oblivious linear evaluation from ring learning with errors," Cryptology ePrint Archive, Report 2020/685, 2020, https://eprint.iacr.org/2020/685.

[28] C. de Saint Guilhem, L. De Meyer, E. Orsini, and N. P. Smart, "BBQ: Using AES in picnic signatures," in *Annual International Workshop on Selected Areas in Cryptography (SAC) 2019*, ser. LNCS. Springer, 2019, pp. 669–692.

[29] S. Dittmer, Y. Ishai, and R. Ostrovsky, "Line-point zero knowledge and its applications," Cryptology ePrint Archive, Report 2020/1446, 2020, https://eprint.iacr.org/2020/1446.

[30] M. F. Esgin, N. K. Nguyen, and G. Seiler, "Practical exact proofs from lattices: New techniques to exploit fully-splitting rings," 2020.

[31] T. K. Frederiksen, J. B. Nielsen, and C. Orlandi, "Privacy-free garbled circuits with applications to efficient zero-knowledge," in *Advances in Cryptology—Eurocrypt 2015, Part II*, ser. LNCS, vol. 9057. Springer, 2015, pp. 191–219.

[32] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *Advances in Cryptology—Eurocrypt 2013*, ser. LNCS. Springer, 2013, pp. 626–645.

[33] I. Giacomelli, J. Madsen, and C. Orlandi, "ZKBoo: Faster zero-knowledge for Boolean circuits," in *USENIX Security Symposium 2016*. USENIX Association, 2016, pp. 1069–1083.

[34] N. Gilboa, "Two party RSA key generation," in *Advances in Cryptology—Crypto 1999*, ser. LNCS, vol. 1666. Springer, 1999, pp. 116–129.

[35] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, no. 4, pp. 792–807, Oct. 1986.

[36] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *19th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 1987, pp. 218–229.

[37] ——, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *J. ACM*, vol. 38, no. 3, pp. 691–729, 1991.

[38] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," in *40th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 2008, pp. 113–122.

[39] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems (extended abstract)," in *17th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 1985, pp. 291–304.

[40] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *Advances in Cryptology—Asiacrypt 2010*, ser. LNCS. Springer, 2010, pp. 321–340.

[41] C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez, "TinyKeys: A new approach to efficient multi-party computation," in *Advances in Cryptology—Crypto 2018, Part III*, ser. LNCS, vol. 10993. Springer, 2018, pp. 3–33.

[42] D. Heath and V. Kolesnikov, "Stacked garbling for disjunctive zero-knowledge proofs," in *Advances in Cryptology—Eurocrypt 2020, Part III*, ser. LNCS, vol. 12107. Springer, 2020, pp. 569–598.

[43] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security Symposium 2011*. USENIX Association, 2011.

[44] R. Impagliazzo, L. A. Levin, and M. Luby, "Pseudo-random generation from one-way functions (extended abstracts)," in *21st Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 1989, pp. 12–24.

[45] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *Advances in Cryptology—Crypto 2003*, ser. LNCS, vol. 2729. Springer, 2003, pp. 145–161.

[46] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Zero-knowledge from secure multiparty computation," in *39th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 2007, pp. 21–30.

[47] M. Jawurek, F. Kerschbaum, and C. Orlandi, "Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently," in *ACM Conf. on Computer and Communications Security (CCS) 2013*. ACM Press, 2013, pp. 955–966.

[48] J. Katz, V. Kolesnikov, and X. Wang, "Improved non-interactive zero knowledge with applications to post-quantum signatures," in *ACM Conf. on Computer and Communications Security (CCS) 2018*. ACM Press, 2018, pp. 525–537.

[49] M. Keller, E. Orsini, and P. Scholl, "Actively secure OT extension with optimal overhead," in *Advances in Cryptology—Crypto 2015, Part I*, ser. LNCS, vol. 9215. Springer, 2015, pp. 724–741.

[50] ——, "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer," in *ACM Conf. on Computer and Communications Security (CCS) 2016*. ACM Press, 2016, pp. 830–842.

[51] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *Advances in Cryptology—Crypto 2012*, ser. LNCS, vol. 7417. Springer, 2012, pp. 681–700.

[52] J. B. Nielsen and C. Orlandi, "LEGO for two-party secure computation," in *6th Theory of Cryptography Conference—TCC 2009*, ser. LNCS, vol. 5444. Springer, 2009, pp. 368–386.

[53] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, "Distributed vector-OLE: Improved constructions and implementation," in *ACM Conf. on Computer and Communications Security (CCS) 2019*. ACM Press, 2019, pp. 1055–1072.

[54] S. Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," in *Advances in Cryptology—Crypto 2020, Part III*, ser. LNCS. Springer, 2020, pp. 704–737.

[55] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *IEEE Symp. Security and Privacy 2018*. IEEE, 2018, pp. 926–943.

[56] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and A. Shelat, "Blind certificate authorities," in *IEEE Symp. Security and Privacy 2019*. IEEE, 2019, pp. 1015–1032.

[57] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.

[58] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Advances in Cryptology—Crypto 2019, Part III*, ser. LNCS, vol. 11694. Springer, 2019, pp. 733–764.

[59] K. Yang, X. Wang, and J. Zhang, "More efficient MPC from improved triple generation and authenticated garbling," in *ACM Conf. on Computer and Communications Security (CCS) 2020*. ACM Press, 2020, pp. 1627–1646.

[60] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated OT with small communication," in *ACM Conf. on Computer and Communications Security (CCS) 2020*. ACM Press, 2020, pp. 1607–1626.

[61] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole - reducing data transfer in garbled circuits using half gates," in *Advances in Cryptology—Eurocrypt 2015, Part II*, ser. LNCS, vol. 9057. Springer, 2015, pp. 220–250.

[62] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "DECO: Liberating web data using decentralized oracles for TLS," in *ACM Conf. on Computer and Communications Security (CCS) 2020*. ACM Press, 2020, pp. 1919–1938.

[63] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *IEEE Symp. Security and Privacy 2020*. IEEE, 2020, pp. 859–876.

APPENDIX

A. Other Functionalities

We review the standard ideal functionality for oblivious transfer (OT) in Figure 13.

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

On receiving (m_0, m_1) with $|m_0| = |m_1|$ from a sender $\mathsf{P_A}$ and $b \in \{0,1\}$ from a receiver $\mathsf{P_B}$, send m_b to $\mathsf{P_B}$.

Fig. 13: The OT functionality between $\mathsf{P_A}$ and $\mathsf{P_B}$.

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

Upon receiving V_{A} from $\mathsf{P_A}$ and V_{B} from $\mathsf{P_B}$, send $(V_{\mathsf{A}} \overset{?}{=} V_{\mathsf{B}})$ and V_{A} to $\mathsf{P_B}$, and do:

- If $\mathsf{P_B}$ is honest and $V_{\mathsf{A}} = V_{\mathsf{B}}$, or is corrupted and sends $\mathsf{continue}$, then send $(V_{\mathsf{A}} \overset{?}{=} V_{\mathsf{B}})$ to $\mathsf{P_A}$.
- If $\mathsf{P_B}$ is honest and $V_{\mathsf{A}} \neq V_{\mathsf{B}}$, or is corrupted and sends abort, then send abort to $\mathsf{P_A}$.

Fig. 14: Functionality for a weak equality test.

In Figure 14 we define a functionality $\mathcal{F}_{\mathsf{EQ}}$ implementing a weak equality test that reveals $\mathsf{P_A}$'s input to $\mathsf{P_B}$. This functionality can be easily realized as follows: (1) $\mathsf{P_B}$ commits to V_{B}; (2) $\mathsf{P_A}$ sends V_{A} to $\mathsf{P_B}$; (3) $\mathsf{P_B}$ outputs $(V_{\mathsf{A}} \overset{?}{=} V_{\mathsf{B}})$ and aborts if they are not equal, and then opens V_{B}; (4) if $\mathsf{P_B}$ opened its commitment to a value V_{B}, then $\mathsf{P_A}$ outputs $(V_{\mathsf{A}} \overset{?}{=} V_{\mathsf{B}})$; otherwise it aborts. UC commitments can be realized efficiently in the random-oracle model.

B. Methods for Batch Checking

We describe two approaches for batch checking of authenticated values. The first relies on a cryptographic hash function H. Specifically, $\mathsf{P_A}$ sends (in addition to the values $x_1, \ldots, x_\ell$ themselves) a digest $h := \mathsf{H}(\mathsf{M}[x_1], \ldots, \mathsf{M}[x_\ell])$ of all the MAC tags; $\mathsf{P_B}$ then checks that $h \overset{?}{=} \mathsf{H}(\mathsf{K}[x_1] + \Delta \cdot x_1, \ldots, \mathsf{K}[x_\ell] + \Delta \cdot x_\ell)$. Modeling H as a random oracle with 2κ-bit output, it is not hard to see that the soundness error (i.e., the probability that $\mathsf{P_A}$ can successfully cheat about *any* value) is upper bounded by $(q_{\mathsf{H}}^2 + 1)/2^{2\kappa} + 1/p^r$, where q_{H} denotes the number of queries that $\mathsf{P_A}$ makes to H. The communication overhead is only 2κ bits, independent of ℓ.

The second approach, which is information theoretic, works as follows:

1) $\mathsf{P_A}$ sends $x_1, \ldots, x_\ell \in \mathbb{F}_p$ to $\mathsf{P_B}$.
2) $\mathsf{P_B}$ picks uniform $\chi_1, \ldots, \chi_\ell \in \mathbb{F}_{p^r}$ and sends them to $\mathsf{P_A}$.
3) $\mathsf{P_A}$ computes $\mathsf{M}[x] := \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i]$, and sends it to $\mathsf{P_B}$.
4) $\mathsf{P_B}$ computes $x := \sum_{i=1}^{\ell} \chi_i \cdot x_i \in \mathbb{F}_{p^r}$ and $\mathsf{K}[x] := \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{K}[x_i] \in \mathbb{F}_{p^r}$. It accepts the opened values if and only if $\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x$.

The soundness error of this approach is given by Lemma 2.

Lemma 2. *Let $x_1, \ldots, x_\ell \in \mathbb{F}_p$ and $\mathsf{M}[x_1], \ldots, \mathsf{M}[x_\ell] \in \mathbb{F}_{p^r}$ be arbitrary values known to $\mathsf{P_A}$, and let Δ and*

Protocol $\Pi^{p,r}_{\mathsf{COPEe}}$

Let $m = \lceil \log p \rceil$ and PRF be a keyed function.

Initialize: This initialization procedure is executed only once.

1) For $i \in [rm]$, $\mathsf{P_A}$ samples $K_0^i, K_1^i \leftarrow \{0,1\}^\kappa$. $\mathsf{P_B}$ samples $\Delta \leftarrow \mathbb{F}_{p^r}$ and lets $\mathbf{\Delta}_B = (\Delta_1, \ldots, \Delta_{rm}) \in \{0,1\}^{rm}$ be its bit-decomposition.
2) For $i \in [rm]$, $\mathsf{P_A}$ sends (K_0^i, K_1^i) to $\mathcal{F}_{\mathsf{OT}}$ and $\mathsf{P_B}$ sends $\Delta_i \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$, which returns $K^i_{\Delta_i}$ to $\mathsf{P_B}$.

Extend: This procedure can be executed multiple times. For the jth input $u \in \mathbb{F}_p$ from $\mathsf{P_A}$, the parties execute the following:

3) For $i \in [rm]$, do the following in parallel:
   a) $\mathsf{P_A}$ sets $w_0^i := \mathsf{PRF}(K_0^i, j)$ and $w_1^i := \mathsf{PRF}(K_1^i, j)$ with $w_0^i, w_1^i \in \mathbb{F}_p$; $\mathsf{P_B}$ computes $w^i_{\Delta_i} := \mathsf{PRF}(K^i_{\Delta_i}, j)$.
   b) $\mathsf{P_A}$ sends $\tau^i := w_0^i - w_1^i - u \in \mathbb{F}_p$ to $\mathsf{P_B}$.
   c) $\mathsf{P_B}$ computes $v^i := w^i_{\Delta_i} + \Delta_i \cdot \tau^i = w_0^i - \Delta_i \cdot u \in \mathbb{F}_p$.
4) Let $\boldsymbol{v} = (v^1, \ldots, v^{rm})$ and $\boldsymbol{w} = (w_0^1, \ldots, w_0^{rm})$ such that $\boldsymbol{w} = \boldsymbol{v} + u \cdot \mathbf{\Delta}_B \in \mathbb{F}_p^{rm}$.
5) $\mathsf{P_A}$ outputs $w = \langle \boldsymbol{g}, \boldsymbol{w} \rangle \in \mathbb{F}_{p^r}$ and $\mathsf{P_B}$ outputs $v = \langle \boldsymbol{g}, \boldsymbol{v} \rangle \in \mathbb{F}_{p^r}$, where $w = v + \Delta \cdot u \in \mathbb{F}_{p^r}$.

Fig. 15: COPEe protocol in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.

$\{\mathsf{K}[x_i] = \mathsf{M}[x_i] - \Delta \cdot x_i\}_{i=1}^{\ell}$*, for uniform* $\Delta \in \mathbb{F}_{p^r}$*, be given to* $\mathsf{P_B}$*. The probability that* $\mathsf{P_A}$ *can successfully open values* $(x_1', \ldots, x_\ell') \neq (x_1, \ldots, x_\ell)$ *to* $\mathsf{P_B}$ *is at most* $2/p^r$*.*

Proof. Fix $(x_1', \ldots, x_\ell') \neq (x_1, \ldots, x_\ell)$ sent by $\mathsf{P_A}$ in the first step. If we let $\omega \overset{\text{def}}{=} \sum_{i=1}^{\ell} \chi_i \cdot (x_i' - x_i)$, then the probability (over uniform choice of $\{\chi_i\}$) that $\omega = 0$ is at most $1/p^r$.

Assume $\omega \neq 0$. If $\mathsf{P_A}$ sends $\mathsf{M} \in \mathbb{F}_{p^r}$, $\mathsf{P_B}$ accepts only if

$$\begin{aligned}
\mathsf{M} &= \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{K}[x_i] + \Delta \cdot \sum_{i=1}^{\ell} \cdot \chi_i \cdot x_i' \\
&= \sum_{i=1}^{\ell} \chi_i \cdot (\mathsf{M}[x_i] - \Delta \cdot x_i) + \Delta \cdot \sum_{i=1}^{\ell} \chi_i \cdot x_i' \\
&= \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i] + \Delta \cdot \omega.
\end{aligned}$$

Everything in the final expression is fixed except for Δ. Moreover, $\mathsf{P_A}$ succeeds iff $\Delta = \omega^{-1} \cdot (\mathsf{M} - \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i])$, which occurs with probability $1/p^r$. □

We can make the second approach *non-interactive*, using the Fiat-Shamir heuristic in the random-oracle model, by computing the coefficients $\{\chi_i\}$ as the output of a hash function H evaluated on the values $\{x_i\}$ sent by $\mathsf{P_A}$ in the first step. Adapting the above proof, one can show that this has soundness error at most $(q_{\mathsf{H}} + 2)/p^r$.

C. Construction of COPEe

In Figure 15, we present a protocol $\Pi^{p,r}_{\mathsf{COPEe}}$ that UC-realizes $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model. This protocol follows the construction of Keller et al. [50], which is in turn based on the IKNP OT-extension protocol [45] and Gilboa's approach [34] for oblivious product evaluation. The main difference from prior work is that we support the subfield case.

Lemma 3. *If* PRF *is a pseudorandom function, then* $\Pi^{p,r}_{\mathsf{COPEe}}$ *UC-realizes* $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ *in the* $\mathcal{F}_{\mathsf{OT}}$*-hybrid model.*

The proof of Lemma 3 can be straightforwardly obtained by following the proof of Keller et al. [50], and is thus omitted.

D. Proof of Theorem 3

We first consider the case of a malicious $\mathsf{P_A}$ and then consider the case of a malicious $\mathsf{P_B}$. In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ that runs the PPT adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, and $\mathcal{F}_{\mathsf{EQ}}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

Malicious $\mathsf{P_A}$. Every time the extend procedure is run (on input n), $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ and records the values (a, c) that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. When $\mathcal{A}$ sends the message $a' \in \mathbb{F}_p$, then $\mathcal{S}$ sets $\beta := a' + a \in \mathbb{F}_p$ and $\delta := c$.
2) For $i \in [1, h)$, $\mathcal{S}$ samples $K^i \leftarrow \{0,1\}^\kappa$; it also samples $K^h \leftarrow \mathbb{F}_{p^r}$. Then for $i \in [h]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ by receiving $\bar{\alpha}_i \in \{0,1\}$ from $\mathcal{A}$, and returning $K^i_{\bar{\alpha}_i} := K^i$ to $\mathcal{A}$. It sets $\alpha := \alpha_1 \cdots \alpha_h$ and defines $\boldsymbol{u} \in \mathbb{F}_p^n$ as the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] := \beta$. Next, $\mathcal{S}$ computes $\{v_j\}_{j \neq \alpha} := \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [h]})$.
3) $\mathcal{S}$ picks $d \leftarrow \mathbb{F}_{p^r}$ and sends it to $\mathcal{A}$. Then, $\mathcal{S}$ defines $\boldsymbol{w}$ as the vector of length n with $\boldsymbol{w}[i] := v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] := \delta - \big(d + \sum_{i \neq \alpha} \boldsymbol{w}[i]\big)$.
4) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ by recording $(\boldsymbol{x}, \boldsymbol{z})$ from $\mathcal{A}$.
5) $\mathcal{S}$ receives $\{\chi_i\}_{i \in [0,n)}$ and $\boldsymbol{x}^* \in \mathbb{F}_p^r$ from $\mathcal{A}$, and sets $\boldsymbol{x}' := \boldsymbol{x}^* + \boldsymbol{x} \in \mathbb{F}_p^r$ and $x' := \sum_{i=0}^{r-1} \boldsymbol{x}'[i] \cdot \mathsf{X}^i$.
6) $\mathcal{S}$ records $V_{\mathsf{A}} \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{EQ}}$. It then computes $V_{\mathsf{A}}' := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and does:
   - If $x' = \beta \cdot \chi_\alpha$, then $\mathcal{S}$ checks whether $V_{\mathsf{A}} = V_{\mathsf{A}}'$. If so, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.
   - Otherwise, $\mathcal{S}$ computes $\Delta' := (V_{\mathsf{A}}' - V_{\mathsf{A}})/(\beta \cdot \chi_\alpha - x') \in \mathbb{F}_{p^r}$ and sends a global-key query (guess, Δ') to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ returns success, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.
7) Whenever $\mathcal{A}$ sends a global-key query $(\text{guess}, \tilde{\Delta})$ to functionality $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, $\mathcal{S}$ forwards the query to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and returns the answer to $\mathcal{A}$. If the answer is abort, $\mathcal{S}$ aborts.

In the above simulation, if $\mathcal{A}$ succeeds to guess Δ, then $\mathcal{S}$ simulates the $\mathcal{A}$'s view using Δ without making any further global-key query to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$.

We claim that the joint distribution of the view of $\mathcal{A}$ and the output of the honest $\mathsf{P_B}$ in the ideal-world execution above is computationally indistinguishable from their distribution in the real-world execution. By the standard analysis of the GGM construction, it is not hard to see that d and the $\{K^i_{\bar{\alpha}_i}\}$ sent to $\mathcal{A}$ in the above simulation, as well as the vector $\boldsymbol{v}$ that would be output by $\mathsf{P_B}$ when it does not abort, are computationally indistinguishable from the corresponding values in the real protocol execution. It thus only remains to analyze steps 4–6, which determine whether $\mathsf{P_B}$ aborts.

Let $\beta = a' + a$, $\boldsymbol{x}' = \boldsymbol{x}^* + \boldsymbol{x}$, and $x' = \sum_{i=0}^{r-1} \boldsymbol{x}'[i] \cdot \mathsf{X}^i$, as above. (Note that $a', a, \boldsymbol{x}^*, \boldsymbol{x}$ are well-defined in the real-world execution as well.) In the real-world execution, $\mathsf{P_B}$ computes

$$
\begin{aligned}
V_{\mathsf{B}} &= \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i \\
&= \sum_{i \neq \alpha} \chi_i \cdot \boldsymbol{v}[i] + \chi_\alpha \cdot \boldsymbol{v}[\alpha] - \sum_{i=0}^{r-1} (\boldsymbol{z}[i] - \Delta \cdot \boldsymbol{x}'[i]) \cdot \mathsf{X}^i \\
&= \sum_{i \neq \alpha} \chi_i \cdot \boldsymbol{v}[i] + \chi_\alpha \cdot (\delta - \Delta \cdot \beta - d - \sum_{i \neq \alpha} \boldsymbol{v}[i]) \\
&\quad - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i + \Delta \cdot x' \\
&= \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i - \Delta \cdot (\beta \cdot \chi_\alpha - x') \\
&= V_{\mathsf{A}}' - \Delta \cdot (\beta \cdot \chi_\alpha - x').
\end{aligned}
$$

where $\boldsymbol{w}$ and V_{A}' are defined as in the description of $\mathcal{S}$ above. Say that $\mathcal{A}$ sends V_{A} to $\mathcal{F}_{\mathsf{EQ}}$. If $x' = \beta \cdot \chi_\alpha$ (as will be the case when $\mathcal{A}$ behaves honestly), then $\mathcal{F}_{\mathsf{EQ}}$ returns true iff $V_{\mathsf{A}} = V_{\mathsf{A}}'$. Otherwise, $\mathcal{F}_{\mathsf{EQ}}$ returns true iff $\Delta = (V_{\mathsf{A}}' - V_{\mathsf{A}})/(\beta \cdot \chi_\alpha - x')$. We thus see that the ideal-world behavior of $\mathcal{F}_{\mathsf{EQ}}$ matches what would occur in the real world.

Malicious $\mathsf{P_B}$. Simulator $\mathcal{S}$ interacts with $\mathcal{A}$ as follows. First, $\mathcal{S}$ simulates the initialization step by recording the global key $\Delta \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. Then, every time the extend procedure is executed (on input n), $\mathcal{S}$ does:

1) $\mathcal{S}$ records $b \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. Then $\mathcal{S}$ samples $a' \leftarrow \mathbb{F}_p$ and sends it to $\mathcal{A}$. Next, $\mathcal{S}$ computes $\gamma := b - \Delta \cdot a'$, and then samples $\beta \leftarrow \mathbb{F}_p^*$ and sets $\delta := \gamma + \Delta \cdot \beta$.
2) $\mathcal{S}$ records the values $\{(K_0^i, K_1^i)\}_{i \in [h]}$ sent to $\mathcal{F}_{\mathsf{OT}}$ by $\mathcal{A}$.
3) $\mathcal{S}$ receives $d \in \mathbb{F}_{p^r}$ from $\mathcal{A}$. Then, for each $\alpha \in [0, n)$, it computes a vector $\boldsymbol{w}_\alpha$ as follows:
   a) Execute $\{v_j^\alpha\}_{j \neq \alpha} := \mathsf{GGM}'(\alpha, \{K_{\bar{\alpha}_i}^i\}_{i \in [h]})$ and set $\boldsymbol{w}_\alpha[i] = v_i^\alpha$ for $i \neq \alpha$.
   b) Compute $\boldsymbol{w}_\alpha[\alpha] := \delta - (d + \sum_{i \neq \alpha} \boldsymbol{w}_\alpha[i])$.
4) $\mathcal{S}$ records the vector $\boldsymbol{y}^*$ sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$.
5) $\mathcal{S}$ samples $\chi_i \leftarrow \mathbb{F}_{p^r}$ for $i \in [0, n)$ and $\boldsymbol{x}^* \leftarrow \mathbb{F}_p^r$, and sends them to $\mathcal{A}$. Then $\mathcal{S}$ computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^*$.
6) $\mathcal{S}$ computes $Y := \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. It then records V_{B} sent to $\mathcal{F}_{\mathsf{EQ}}$ by $\mathcal{A}$. Next, $\mathcal{S}$ computes a set $I \subseteq [0, n)$ as follows:
   a) For $\alpha \in [0, n)$, compute $V_{\mathsf{A}}^\alpha := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}_\alpha[i] - \Delta \cdot \beta \cdot \chi_\alpha - Y$.
   b) Define $I := \{\alpha \in [0, n) \mid V_{\mathsf{A}}^\alpha = V_{\mathsf{B}}\}$.

   $\mathcal{S}$ sends I to $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$; if it returns abort, $\mathcal{S}$ picks $\tilde{\alpha} \leftarrow [0, n) \backslash I$, sends $\left(\mathsf{false}, V_{\mathsf{A}}^{\tilde{\alpha}}\right)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{EQ}}$, and then aborts. Otherwise, $\mathcal{S}$ sends $(\mathsf{true}, V_{\mathsf{B}})$ to $\mathcal{A}$.
7) $\mathcal{S}$ chooses an arbitrary $\alpha \in I$ and computes $\boldsymbol{v}$ as follows:
   a) Set $\boldsymbol{v}[i] := \boldsymbol{w}_\alpha[i]$ for $i \in [0, n), i \neq \alpha$.
   b) Set $\boldsymbol{v}[\alpha] := \gamma - d - \sum_{i \neq \alpha} \boldsymbol{v}[i]$.

   $\mathcal{S}$ sends $\boldsymbol{v}$ to $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ and outputs whatever $\mathcal{A}$ outputs.

We first consider the view of adversary $\mathcal{A}$ in the ideal-world execution and the real-world execution. The values a' and $\boldsymbol{x}^*$ simulated by $\mathcal{S}$ have the same distribution as the real values, which are masked by a uniform element/vector output by $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. The set I extracted by $\mathcal{S}$ corresponds to the selective failure attack on the output index α^* of $\mathsf{P_A}$. If $\mathcal{S}$ receives abort from $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$, we have that $\alpha^* \notin I$. In the real protocol execution, if $V_{\mathsf{B}} \neq V_{\mathsf{A}}^{\alpha^*}$, then $\mathsf{P_A}$ aborts. By previous considerations, this is equivalent to $\alpha^* \notin I$. Therefore, $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ aborts if and only if the real protocol execution aborts. For an honest $\mathsf{P_A}$, the index $\alpha^* \in [0, n)$ is sampled uniformly in both the real-world execution and the ideal-world execution. If receiving abort from $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$, then $\mathcal{S}$ needs to send false along with an element $V_{\mathsf{A}}^{\tilde{\alpha}} \neq V_{\mathsf{B}}$ to $\mathcal{A}$. Although $\mathcal{S}$ does not know the actual index α^*, it can sample a random index $\tilde{\alpha}$ from the set $[0, n) \backslash I$ and send $V_{\mathsf{A}}^{\tilde{\alpha}}$ to $\mathcal{A}$. In the case of aborting, this simulation is perfect, since $\mathcal{Z}$ cannot obtain the output of $\mathsf{P_A}$ due to aborting, and the dummy index $\tilde{\alpha}$ has the same distribution as the actual index α^* under the condition that I is an incorrect guess.

Overall, we have that the adversary's view is perfectly indistinguishable between the real-world execution and the ideal-world execution. Below, we prove that except with probability $1/p^r$, the distribution of $\mathsf{P_A}$'s output in the real-world execution is the same as that in the ideal-world execution. It is easy to see that the output vector $\boldsymbol{u}^*$ that is 0 everywhere except that $\boldsymbol{u}^*[\alpha^*] = \beta^*$ in the ideal-world execution and the real-world execution have the same distribution, from the above analysis and that β^* is perfectly hidden. In the following, we focus on proving the indistinguishability of $\boldsymbol{w}^*$ output by $\mathsf{P_A}$ between the ideal-world execution and the real-world execution. Firstly, we prove that the vector $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ computed by $\mathcal{S}$ in the step 7 is unique (i.e., independent of the choice $\alpha \in I$).

Claim 1. *For any $\alpha, \alpha' \in [0, n)$, let $\boldsymbol{v}_\alpha, \boldsymbol{v}_{\alpha'}$ be the vectors computed by $\mathcal{S}$ with α, α' following the step 7, then we have*

$$
\Pr\left\{ \boldsymbol{v}_\alpha \neq \boldsymbol{v}_{\alpha'} \;\middle|\; V_{\mathsf{A}}^\alpha = V_{\mathsf{A}}^{\alpha'} \right\} \leq \frac{1}{p^r}.
$$

The proof of the claim described as above is similar to that in prior work [60], and is not included due to the space limitations; it is available for reviewers upon request.

Let $\boldsymbol{w}^*, \boldsymbol{u}^*$ be the output of $\mathsf{P_A}$ and $\boldsymbol{v}$ be the input from $\mathcal{S}$ (or $\mathsf{P_B}$). It is obvious that $\boldsymbol{w}^* = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}^*$ in the ideal-world execution. Now we look at the real-world execution. We define a vector $\boldsymbol{v}^*$ as $\boldsymbol{v}^*[i] = \boldsymbol{w}_{\alpha^*}[i]$ for $i \neq \alpha^*$ and $\boldsymbol{v}^*[\alpha^*] = \gamma - d - \sum_{i \neq \alpha^*} \boldsymbol{v}^*[i]$, where recall that α^* is the output index of $\mathsf{P_A}$. From $\boldsymbol{w}_{\alpha^*}[\alpha^*] = \gamma + \Delta \cdot \beta^* - (d + \sum_{i \neq \alpha^*} \boldsymbol{w}_{\alpha^*}[i])$, we have that $\boldsymbol{w}_{\alpha^*}[\alpha^*] = \boldsymbol{v}^*[\alpha^*] + \Delta \cdot \beta^*$. Therefore, we obtain that $\boldsymbol{w}^* = \boldsymbol{v}^* + \Delta \cdot \boldsymbol{u}^*$ where $\boldsymbol{w}^* = \boldsymbol{w}_{\alpha^*}$. Note that $\boldsymbol{v}^*$ in both the ideal-world execution and the real-world execution are defined in the identical way, and thus have the same distribution. Based on Claim 1, we know that in the ideal-world execution, $\boldsymbol{v}^*$ is indistinguishable from $\boldsymbol{v}$ computed by

$\mathcal{S}$, except with probability at most $1/p^r$. Therefore $\boldsymbol{v}$ in the ideal-world execution is indistinguishable from $\boldsymbol{v}^*$ in the real-world execution, which implies the indistinguishability of the output of $\mathsf{P_A}$ in the ideal world and the real world.

E. Proof of Theorem 4 and Protocol Optimizations

Proof. We first consider the case of a malicious $\mathsf{P_A}$ and then consider the case of a malicious $\mathsf{P_B}$. In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ that runs the adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ and $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and $\mathcal{Z}$.

Malicious $\mathsf{P_A}$. $\mathcal{S}$ records the vectors $(\boldsymbol{u}, \boldsymbol{w}) \in \mathbb{F}_p^k \times \mathbb{F}_{p^r}^k$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ during initialization. Then in each iteration, $\mathcal{S}$ runs as follows:

1) For $i \in [t]$, $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and receives the value $\boldsymbol{e}_i \in \mathbb{F}_p^m$ (with at most one nonzero entry) and $\boldsymbol{c}_i \in \mathbb{F}_{p^r}^m$ from $\mathcal{A}$; it then defines $\boldsymbol{e} := (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^n$ and $\boldsymbol{c} := (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^n$.
2) $\mathcal{S}$ computes $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e} \in \mathbb{F}_p^n$ and $\boldsymbol{z} := \boldsymbol{w} \cdot \mathbf{A} + \boldsymbol{c} \in \mathbb{F}_{p^r}^n$, and sends $\boldsymbol{x}[k : n) \in \mathbb{F}_p^\ell$ and $\boldsymbol{z}[k : n) \in \mathbb{F}_{p^r}^\ell$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. It also locally updates $\boldsymbol{u} := \boldsymbol{x}[0 : k) \in \mathbb{F}_p^k$ and $\boldsymbol{w} := \boldsymbol{z}[0 : k) \in \mathbb{F}_{p^r}^k$ for the next iteration.
3) If $\mathcal{A}$ ever makes a global key query Δ' to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$, then $\mathcal{S}$ forwards that query to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ responds with abort, $\mathcal{S}$ aborts; otherwise, it continues.

It is easy to see that the simulation provided by $\mathcal{S}$ is perfect.

Malicious $\mathsf{P_B}$. $\mathcal{S}$ runs $\mathcal{G}(1^k, 1^n, p)$ to generate $\mathbf{A} \in \mathbb{F}_p^{k \times n}$. During initialization, $\mathcal{S}$ records the values $\Delta \in \mathbb{F}_{p^r}$ and $\boldsymbol{v} \in \mathbb{F}_{p^r}^k$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, and sends Δ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. Then in each iteration, $\mathcal{S}$ runs as follows:

1) For $i \in [t]$, $\mathcal{S}$ receives the value $\boldsymbol{b}_i \in \mathbb{F}_{p^r}^m$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$; it sets $\boldsymbol{b} := (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^n$.
2) For $i \in [t]$, $\mathcal{S}$ receives the set $I_i \subseteq [0, m)$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Then $\mathcal{S}$ samples $\boldsymbol{e} \leftarrow \mathcal{D}_{n,t}$ and defines $\{\alpha_1, \ldots, \alpha_t\}$ to be the nonzero entries of $\boldsymbol{e}$. If $\alpha_i \bmod m \in I_i$ for all i, then $\mathcal{S}$ continues; otherwise, it aborts.
3) $\mathcal{S}$ computes $\boldsymbol{y} := \boldsymbol{v} \cdot \mathbf{A} + \boldsymbol{b} \in \mathbb{F}_{p^r}^n$, and sends $\boldsymbol{y}[k : n) \in \mathbb{F}_{p^r}^\ell$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. It also locally updates $\boldsymbol{v} := \boldsymbol{y}[0 : k) \in \mathbb{F}_{p^r}^k$ for the next iteration.

The view of $\mathcal{A}$ is simulated perfectly, and in both the ideal-world simulation and the ideal-world execution of the protocol the output $(\boldsymbol{s}, \mathsf{M}[\boldsymbol{s}])$ of $\mathsf{P_A}$ satisfies $\boldsymbol{y}[k, n) = \mathsf{M}[\boldsymbol{s}] - \Delta \cdot \boldsymbol{s}$. The difference is that in the ideal world $\boldsymbol{s}$ is uniform, whereas in the real world $\boldsymbol{s} = \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e}$ for a uniform vector $\boldsymbol{u}$. It is not hard to see that this difference is undetectable if the $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ assumption holds. □

Optimizations. In each iteration of the extend procedure, protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ makes t calls to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ is instantiated by protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$ from Section IV-B, and we use the optimization described at the end of that section, the t calls to $\Pi^{p,r}_{\mathsf{spsVOLE}}$ require only $t + r$ calls to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$.

Moreover, we can push all the calls to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ into the initialization phase, so that the extend procedure does not invoke $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ at all. Specifically, if we make $n_0 = k + t + r$ calls to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ during initialization, we can run the extend procedure without any additional call to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. Each time the extend procedure is run, we reserve n_0 of the sVOLE correlations that are produced for the following iteration, and output $n - n_0$ "usable" sVOLE correlations.

We can further optimize the generation of the initial set of n_0 sVOLE correlations during initialization. Let (k_0, n_0, t_0) be another set of LPN parameters. (Note that $n_0 \ll n$, so we can take $k_0 \ll k$ and $t_0 \approx t$ while achieving security comparable to what is achieved for the LPN parameters (n, k, t).) We then make $n_0' = k_0 + t_0 + r$ calls to the base-sVOLE protocol described in Section IV-A to generate that number of sVOLE correlations, after which we run the extend procedure of $\Pi^{p,r}_{\mathsf{sVOLE}}$ once to obtain n_0 sVOLE correlations.