Dishonest Majority Multi-Verifier Zero-Knowledge Proofs for Any Constant Fraction of Corrupted Verifiers

Daniel Escudero,[1] Antigoni Polychroniadou,[1] Yifan Song[2,3] and Chenkai Weng[4]

[1] J.P. Morgan AI Research & J.P. Morgan AlgoCRYPT CoE, NY, USA
[2] Tsinghua University, Beijing, China
[3] Shanghai Qi Zhi Institute, Shanghai, China
[4] Arizona State University, AZ, USA

Abstract. In this work we study the efficiency of Zero-Knowledge (ZK) arguments of knowledge, particularly exploring Multi-Verifier ZK (MVZK) protocols as a midway point between Non-Interactive ZK and Designated-Verifier ZK, offering versatile applications across various domains. We introduce a new MVZK protocol designed for the preprocessing model, allowing any constant fraction of verifiers to be corrupted, potentially colluding with the prover. Our contributions include the first MVZK over rings. Unlike recent prior works on fields in the dishonest majority case, our protocol demonstrates communication complexity independent of the number of verifiers, contrasting the linear complexity of previous approaches. This key advancement ensures improved scalability and efficiency. We provide an end-to-end implementation of our protocol. The benchmark shows that it achieves a throughput of 1.47 million gates per second for 64 verifiers with 50% corruption, and 0.88 million gates per second with 75% corruption.

1 Introduction

A Zero-Knowledge (ZK) proof is a protocol between a *prover* and a *verifier* which enables the first to convince the second that a given statement is true, without leaking anything beyond the validity of the statement. For instance, let us say Alice wants to prove to Bob that she knows a specific password to access a restricted area without actually revealing the password to Bob. Using a ZK proof, Alice can demonstrate her knowledge of the password without disclosing the password itself. ZK proofs exhibit a wide array of applications spanning various domains. In Cryptocurrencies and Blockchain Technology, ZK proofs verify transactions while concealing sender, recipient, and transaction amounts [BS+14; Bün+20], preserving privacy in decentralized systems. They also feature prominently in Privacy-Preserving Technologies, facilitating anonymous credentials or digital signatures [Boo+23] by validating properties without disclosing actual information. In cloud computing and verifiable computation, ZK proofs ensure data integrity and confidentiality, enabling secure computations, including machine learning tasks [Xin+23], without exposing sensitive information. Furthermore, from seminal works [GMW87] to recent financial applications [Pol+23], ZK proofs establish malicious security in secure multiparty computation, proving their worth against malicious adversaries.

Recent development of IOP-based zk-SNARKs demonstrate efficient proof generation [Gol+23; XZS22; WHV24]. However, the scalability remains an issue for these schemes and all other non-interactive ZK proofs. Specifically, the memory overhead grows with the increase of the statement size. In order to generate proofs for complicated statements such as zkEVM or zkML, prior benchmarks often require machines with at least hundreds of GBs of memory. This bottleneck prevents these schemes from being deployed in the commodity hardware, and incurs high financial cost for ZK applications.

On the other hand, recent development of designated verifier ZK proofs shows high prover efficiency and scalability [Wen+21b; Yan+21; Bau+21; DIO20]. By allowing a constant rounds of communication, they enable the streaming of proofs and result in throughput of tens of million gates per second. More importantly, the memory overhead of these schemes ranges from several hundred MBs to a few GBs no matter how large the statement is. Their applications include the proof of large-scale deep neural network inferences [Wen+21a] and database SQL query processing [Li+23]. However, they only allow a single designated verifier and thus are not suitable for proving to a group of verifiers. For instance, private aggregation systems require a user to validate its input

to multiple servers using ZK proofs [CGB17; Add+22; Rat+23]. Also, blockchain oracles verify sensitive off-chain user data by verifying the ZK proofs from users [Zha+20].

In this context, multi-verifier zero-knowledge (MVZK) proofs are developed [YW22; Bau+22a; AKP22; Zho+23], which allow for multiple verifiers to jointly verify a statement of a prover while maintaining its privacy. The soundness guarantee of MVZK requires that the honest verifiers should not be convinced even if a corrupted prover colludes with a set of corrupted verifiers. The zero-knowledge property requires that the corrupted verifiers learn no information about an honest prover's witness. Crucially, regarding performance, MVZK proofs inherit the prover efficiency and scalability from interactive ZK proofs, which enables large-scale ZKP to a large number of verifiers. Furthermore, they are more communication efficient than executing an interactive proof with each verifier separately, so they offer a good trade-off between the communication complexity of 2-party interactive proofs, and the computational costs of NIZKs. Unfortunately, prior MVZK works either only support a minority of corrupted verifiers [YW22; Bau+22a; AKP22], or they assume the worst case in which all-but-one verifiers are corrupted [Zho+23]. Such a "hard line" imposes quite some restrictions in practice: there are settings where it is not reasonable to assume an honest majority, but still assuming all-but-one party is corrupted is an overkill. It is currently not clear how to improve efficiency of MVZK by assuming more than a single honest party, without reaching the majority threshold.

1.1 Our Contribution

This work bridges the gap between NIZK and DVZK by examining the efficiency of ZK proofs in scenarios where a prover needs to demonstrate validity of a given statement to multiple verifiers. We propose an efficient and scalable MVZK protocol that tolerates *any* constant fraction of corrupted verifiers, which may include the dishonest majority case but does not include the extreme case of only one honest party. Its performance is comparable to the most efficient IOP-based zk-SNARKs and VOLE-based interactive ZK proofs, while scaling from million-size to billion-size circuits with a flat memory usage of a few GBs.

More specifically, we consider the task of MVZK for the setting in which the adversary corrupts t out of the n verifiers, where $t = n(1-\epsilon)$, for any $\epsilon \in (0,1)$. Our scheme also tolerates the prover-verifier collusion. The total prover-to-verifiers communication grows as $O(|C|/\epsilon)$, while the verifiers only communicate $O(n \cdot \mathsf{polylog}(|C|))$ among each other, where $|C|$ is the size of the circuit to be checked. Note that for the case in which $\epsilon = 1/n$, which corresponds to an adversary that corrupts $t = n(1-1/n) = n-1$ parties, prover-verifier communication is linear $O(|C|n)$. However, for $\epsilon = \Theta(1)$, this communication becomes is $O(|C|)$, which is *independent of* n. A constant ϵ translates into the adversary only corrupting a constant fraction of parties, or equivalently, a constant fraction of parties being honest. For example, the adversary may corrupt 80% of the parties, which ensures 20% of the parties are honest. This is an appropriate model for settings with a medium-to-large number of parties, where it may be difficult for the adversary to corrupt all-but-one of the participants, but assuming honest majority may not be viable. We remark that the prior works of [YW22; Bau+22a; AKP22] assume honest majority ($t < n/2$), and [Zho+23] assumed dishonest majority, but where only one party is assumed to be honest, which is too strong and prevents them from achieving certain efficiency features we enjoy.

Our contributions can be summarized as follows:

- **ZK over Rings:** Normally, ZK proofs are given for arithmetic circuits over finite fields due to their amenable algebraic structure, but several other rings, like integers modulo 2^k, may offer benefits in practice (*e.g.* [Dam+19]). Our work is the first one to consider MVZK proofs over more general rings, for any constant fraction of the corrupted verifiers including collusion with the prover. The rings we consider, the so-called Galois rings, include as particular cases the ring from above, as well as finite fields.
- **Asymptotic Analysis:** Our protocol is divided in an offline phase, independent of the witness, and the online phase. For the online phase, our protocol exhibits communication complexity *independent* of n, a crucial factor for scalability and a notable advantage compared to the work of [Zho+23], which requires linear communication in n. Regarding computation, [Zho+23] relies on more computationally expensive techniques, namely homomorphic encryption and generic NIZKs in the preprocessing phase. In contrast, we rely on recent highly optimized techniques for vector oblivious linear evaluation (VOLE).

– **Round Complexity:** The online phase in our approach consists of prover $\mathcal{P}$ sending the proof, followed by two rounds of interaction between verifiers. This complexity aligns with that of the work by [Zho+23].[5]
– **Honest majority Regime:** Given our protocol's flexibility for any ϵ, it can effectively operate in the honest majority setting as well ($\epsilon > 1/2$). While our primary focus remains on the dishonest majority case, this potential improvement in the honest majority regime can be a byproduct of our work. Notably, while the communication complexity of the work in [Bau+22a] in the honest majority case scales linearly with n, in our case, it remains independent of n resulting in significantly improvements. The communication in the work of [YW22], set for $\epsilon \gg 1/2$, does not grow with n (as ours), but as we discuss in Section 5.1 our underlying constants are better.
– **Implementation and Experiments:** We provide an end-to-end implementation of our protocol and benchmark its performance with different number of parties, corruption thresholds and network settings. The results show high efficiency and scalability compared to related works and implementations.

We refer readers to Section 5 for an in-depth comparison covering all the mentioned aspects, including the preprocessing phase.

Applications. Our MVZK proofs boast numerous applications. Below, we delineate some specific key uses. Private aggregation systems such as Prio/Prio+/ELSA [CGB17; Add+22; Rat+23] utilize a network of servers for collecting and aggregating user data. To ensure data accuracy and protect against potential attacks, users are required to validate their data with these servers. This validation process is accomplished through secret-shared non-interactive proof techniques. However, the existing protocol in Prio, while assuming dishonest majority across the verifiers, it assumes that the prover will not collaborate with any verifier, posing limitations. In contrast, our protocol presents a potentially more efficient alternative, even if a user/prover collaborates with a majority of servers. See Section 5.2 for more discussion. Generally speaking, our MVZK proofs can be utilized in lieu of generic Secure Multi-Party Computation involving input or predicate validation, especially in scenarios where a dishonest majority, but a constant amount of honest parties, is expected.

In the blockchain setting, oracles are independent nodes that verify off-chain information. Privacy-preserving oracle solutions such as DECO [Zha+20] enable oracles to verify any computation over private data using zero-knowledge proofs, which prevents oracles from learning users' sensitive information, such as the bank account balance. Currently, the DECO product uses interactive ZKP based on VOLE[6] because the proof involves a circuit of size tens of billions of gates, which is hard to deal with by existing NIZK. But, it only proves to one oracle at a time. Our MVZK protocol enables a blockchain user to prove such a complicated statement to multiple oracles simultaneously without better efficiency.

1.2 Technical Overview

In this section, we provide a brief technical overview and highlight our main technical contributions. Consider a prover $\mathcal{P}$ who holds a witness $\boldsymbol{x}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ who want to verify a statement $C(\boldsymbol{x}) = 0$ for a public arithmetic circuit $C(\cdot)$. For simplicity in the exposition, we assume that both $\boldsymbol{x}$ and the circuit C are defined over a *finite field* $\mathbb{F}$, with the general ring case handled in the formal description of our protocol. Previous multi-verifier zero-knowledge proofs [Bau+22a; YW22; CGB17; AKP22] align with the following paradigm:

1. $\mathcal{P}$ secret-shares the input $\boldsymbol{x}$—under some linear secret-sharing (SS) scheme—towards the n verifiers, which we denote by $[\boldsymbol{x}]$.
2. For every multiplication gate $w_\gamma \leftarrow g_\times(w_\alpha, w_\beta)$, $\mathcal{P}$ secret-shares the output of the gate $[w_\gamma]$. Given that, by using the linearity of the SS scheme the verifiers can locally obtain shares of the output $[w_\gamma]$ of every addition gate gate $w_\gamma \leftarrow g_+(w_\alpha, w_\beta)$, this effectively means the verifiers have shares of *every* wire value in the circuit.
3. What remains is to check that the circuit is computed correctly. This amounts to checking that (1) its output is 0, and (2) the multiplication gates satisfy the correct multiplicative relation

[5] As we will elaborate on, the work of [Zho+23] can have one less round if communication is increased from linear to quadratic $\Omega(n^2)$.
[6] DECO Research Series https://blog.chain.link/deco-introduction/.

$w_\gamma = w_\alpha \cdot w_\beta$. For this, the parties interact in some protocol that takes as input the shares of the wires they hold, and outputs accept/reject.

Previous works use different secret-sharing schemes in conjunction with alternative ways of checking the products. Baum et al. [Bau+22a] propose MVZK protocols with $O(n|C|)$ communication and corruption threshold $t < n/3$; the use of Shamir SS with $t < n/3$ enables the parties to perform one product *non-interactively* while guaranteeing error-detection, meaning corrupted parties cannot modify the underlying secrets. The work of Yang and Wang [YW22] achieves $O(|C|)$ communication (and the prover does not need to participate once the input is secret-shared) with threshold $t < n(1/2 - \epsilon)$ for $\epsilon \in (0, 1/2)$; for this they use packed secret-sharing in conjunction with packed Beaver triples, used for multiplications. Corrigan-Gibbs and Boneh [CGB17] achieve soundness against $t = n - 1$ corrupted verifiers, but it assumes that the prover and verifiers do not collude. Zhou et al. [Zho+23] eliminate this assumption, but it requires $O(n|C|)$ communication and requires heavy MPC computation during its preprocessing phase. In these last two cases, since they are set in the dishonest majority regime, additive secret-sharing with message authentication codes (MACs) is used.

Our goal is to obtain MVZK against dishonest-majority verifiers for $t = n(1 - \epsilon)$, preserving soundness even if $\mathcal{P}$ colludes with t verifiers. We do follow the general template from above, but we aim for a prover-verifier communication complexity of $O(|C|/\epsilon)$ (which for $\epsilon = \Theta(1)$ becomes independent of n), and a verifier-to-verifier communication of $O(n \cdot \mathsf{polylog}|C|)$ (linear in n, but sublinear in $|C|$). In our work, $\mathcal{P}$ will distribute *additive sharings* of the wire values, which we denote by $\langle x \rangle$ for $x \in \mathbb{F}$. However, we cannot let $\mathcal{P}$ simply distribute sharings of each wire since this would cost $O(|C|n)$ communication, while we are aiming at $O(|C|/\epsilon)$. Furthermore, additive secret-sharing is *not* robust, in the sense that corrupted parties can modify their shares and thereby alter the underlying secret, which could enable a corrupted prover who colludes with some corrupted verifiers to pass the verification even if the right multiplicative relations do not hold. As in general-purpose dishonest-majority MPC, this is dealt with via information-theoretic message authentication codes (IT-MACs), which consists of enhancing the sharings $\langle x \rangle$ with $\langle x \cdot \Delta \rangle$, where Δ is a random key which is unknown to the adversary, and is secret-shared as $\langle \Delta \rangle$. We denote this by $[\![x]\!] = (\langle x \rangle, \langle x \cdot \Delta \rangle)$. Letting $\mathcal{P}$ distribute sharings directly would require $\mathcal{P}$ to know the key Δ, which breaks security in the case in which $\mathcal{P}$ is corrupted.

Following the paradigm from previous MVZK works—and as usual in MPC when considering a party who provides input—we do not let $\mathcal{P}$ directly secret-share the extended witness, and instead this party receives a *mask* $\boldsymbol{u}$, which is secret-shared among the parties as $[\![\boldsymbol{u}]\!]$. Then $\mathcal{P}$ distributes the difference between the wire values and $\boldsymbol{u}$ towards the parties, who can *locally* add the shares of $\boldsymbol{u}$, obtaining authenticated shares of all wire values, as required [Bea95]. At this point, two main challenges arise. On the one hand, directly sending the $|C|$ differences to the verifiers is too communication heavy since this would entail $O(n|C|)$ messages, while we are aiming for $O(|C|/\epsilon)$. Secondly, the verifiers need to generate $|C|$ authenticated sharings $[\![\boldsymbol{u}]\!]$ with communication $O(n \cdot \mathsf{polylog}|C|)$, which is sublinear in $|C|$, and furthermore they need to reconstruct this mask to $\mathcal{P}$ with communication $O(|C|/\epsilon)$. These issues are addressed with a combination of pseudorandom correlation generators (PCGs), together with packed secret-sharing, as we elaborate on below.

IT-MACs and Programmable VOLE. First, we observe that correlations of the form $[\![\boldsymbol{u}]\!] = (\langle \boldsymbol{u} \rangle, \langle \boldsymbol{u} \cdot \Delta \rangle)$ can be derived from pairwise vector oblivious linear evaluation (VOLE) correlations. In n-party VOLE, each $\mathcal{V}_i$ holds $\boldsymbol{u}^i$, and for each pair of parties $(\mathcal{V}_i, \mathcal{V}_j)$, they share correlations $\boldsymbol{w}_j^i = \boldsymbol{v}_i^j + \boldsymbol{u}^i \cdot \Delta^j$, where $\mathcal{V}_i$ holds $(\boldsymbol{u}^i, \boldsymbol{w}_j^i)$ and $\mathcal{V}_j$ holds $(\boldsymbol{v}_i^j, \Delta^j)$. If we define $\boldsymbol{u} = \sum_{i=1}^n \boldsymbol{u}^i$ and $\Delta = \sum_{i=1}^n \Delta^i$, we can regard the above correlation as sharings $[\![\boldsymbol{u}]\!] = (\langle \boldsymbol{u} \rangle, \langle \boldsymbol{u} \cdot \Delta \rangle)$.

Now, our construction will require that $\mathcal{P}$ learns $\{\boldsymbol{u}^i\}_{i=1}^n$, where the dimension of these vectors is $m \approx |C|$. A naive solution is to let each verifier $\mathcal{V}_i$ send its $\boldsymbol{u}^i$ to $\mathcal{P}$ after the VOLE executions, but this requires $O(nm) = O(n|C|)$ communication in total, which is too high. Instead, we rely on a variant of VOLE, programmable VOLE [RS22], in which each vector $\boldsymbol{u}^i$ is obtained by applying a pseudorandom function $\mathsf{Expand} : S \to \mathbb{F}^m$ to a *succinct* seed seed^i chosen from a seed space S. With this tool at hand, $\mathcal{P}$ can send each verifier $\mathcal{V}_i$ a seed $\mathsf{seed}^i \in S$, which they use to run programmable VOLE, hence obtaining $[\![\boldsymbol{u}]\!]$. Due to the succinctness of the seeds, these communication costs become

irrelevant for our goals, and furthermore $\mathcal{P}$ can compute the desired $\boldsymbol{u}^i \leftarrow \mathsf{Expand}(\mathsf{seed}^i)$. We point out that $\mathcal{P}$ does not learn Δ or $\boldsymbol{u} \cdot \Delta$, which is crucial for security.

Distributing Extended Witnesses. We denote the secret values that $\mathcal{P}$ distributes at Steps 1-2 as extended witnesses $\boldsymbol{x}$ for $\boldsymbol{x} \in \mathbb{F}^{|C|}$. After the execution of the programmable n-party VOLE, the verifiers hold sharings $[\![\boldsymbol{u}]\!]$, and $\mathcal{P}$ is supposed to send certain information that allows the parties to compute $[\![\boldsymbol{x}]\!]$. In prior works, $\mathcal{P}$ computes the difference $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{u} \in \mathbb{F}^{|C|}$ and broadcasts this to verifiers. Using the linearity of IT-MACs, they can locally compute $[\![\boldsymbol{x}]\!] = [\![\boldsymbol{u}]\!] + \boldsymbol{\delta}$. However, this requires at least total $O(n|C|)$ communication since each verifier needs to receive $O(|C|)$ values (regardless of how the broadcast channel is instantiated). To reduce this to $O(|C|/\epsilon)$, we draw inspiration from SuperPack [Esc+23]—which considers MPC for a similar corruption threshold as us—by making use of *packed secret-sharing* [FY92] to save in communication.

In more detail, let us denote a packed Shamir sharing of degree d as $[\boldsymbol{r}]_d$, with the secret $\boldsymbol{r} \in \mathbb{F}^\sigma$. We will take $\sigma = n \cdot \epsilon$ and the degree d will either be $(n-1)$, which ensures privacy against t corrupted parties since $t = n - n\epsilon$, or $d = \sigma - 1$, which is the minimum degree to store σ secrets. We exploit packed SS as follows. Suppose the parties have a series of shared random vectors $[\boldsymbol{r}_i]_{n-1}$ for $i \in [|C|/\sigma]$, where $\mathcal{P}$ knows $\boldsymbol{r}_i$. To distribute the extended witnesses, $\mathcal{P}$ first arranges it into $m' = |C|/\sigma$ groups defined as $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{m'})$, where each $\boldsymbol{x}_i$ is in $\mathbb{F}^\sigma$. For $i \in [m']$, $\mathcal{P}$ computes $\boldsymbol{\delta}_i = \boldsymbol{x}_i - \boldsymbol{r}_i$ and distributes $[\boldsymbol{\delta}_i]_{\sigma-1}$ to the verifiers. Here a degree-$(\sigma-1)$ packed Shamir sharing is sufficient since we do not need to protect the secrecy of $\boldsymbol{\delta}_i$. As also observed in SuperPack, packed SS can be *locally* converted into additive shares of each secret, so from $[\boldsymbol{\delta}_i]_{\sigma-1}$ and $[\boldsymbol{r}_i]_{n-1}$ all verifiers can locally compute $\{\langle \delta_{i,j} \rangle\}_{j=1}^{\sigma}$ and $\{\langle r_{i,j} \rangle\}_{j=1}^{\sigma}$, which allows them to compute $\langle \tilde{w}_{i,j} \rangle = \langle \delta_{i,j} \rangle + \langle r_{i,j} \rangle$. The total communication from the prover to the verifiers is $(|C|/\sigma) \cdot n = |C| \cdot \epsilon$, as required.

There are two crucial aspects missing for the idea above to work, with the first being that the verifiers should have $[\boldsymbol{r}_i]_{n-1}$ for $i \in [m']$, where $\mathcal{P}$ knows r_i. Here, we use the following pivotal observation from SuperPack: any random additive sharing $\langle u \rangle$ where $\mathcal{V}_i$ holds u^i can be re-interpreted as a random *packed* sharing of some secret $[\boldsymbol{r}]_{n-1}$, by letting $\mathcal{V}_i$'s share of $\boldsymbol{r}$ be u^i itself. Each of the entries in the secret $\boldsymbol{r}$ would be determined by the appropriate linear combination of $(u^1, \ldots, u^n)$ using the corresponding Lagrange coefficients. Now, recall from the previous discussion that, with the help of programmable VOLE, each verifier's share $\boldsymbol{u}^i$ is derived from seed^i that $\mathcal{P}$ sampled and sent to $\mathcal{V}_i$. We think of the verifiers not as having additive sharings $\langle \boldsymbol{u} \rangle$, but actually *packed* sharings $[\boldsymbol{r}_i]_{n-1}$ for $i \in [m']$, which can be converted *locally* from the additive sharings $\langle \boldsymbol{u} \rangle$ based on the aforementioned SuperPack technique. In particular, $\mathcal{P}$ can locally derive $(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_{m'})$ from the distributed seeds, setting up the stage for the verifiers to obtain $\langle \tilde{w}_{i,j} \rangle$ with reduced communication. Finally, we show that from n-party VOLE correlations, all verifiers manage to *locally* obtain $\{\langle r_{i,j} \cdot \Delta \rangle\}_{i \in [m'], j \in [\sigma]}$. Effectively, all verifiers hold $\{[\![r_{i,j}]\!]\}_{i \in [m'], j \in [\sigma]}$ in the end.

The second core issue to be addressed is that what the parties need is not only $\langle \tilde{w}_{i,j} \rangle$, but actually $\{[\![\tilde{w}_{i,j}]\!]\}_{i \in [m'], j \in [\sigma]}$, for what they miss the authentication $\langle \delta_{i,j} \cdot \Delta \rangle$. To this end, we let all verifiers further prepare the following Shamir sharings: $\{[\Delta|_j]_t\}_{j=1}^{\sigma}$, where $[\Delta|_j]_t$ refers to a (non-packed, standard) Shamir sharing where Δ is evaluated at the j-th evaluation point. We note that for $i \in [m']$ and $j \in [\sigma]$, the verifiers can *locally* compute $[\boldsymbol{\delta}_i]_{\sigma-1} \cdot [\Delta|_j]_t$ to obtain a degree-$(n-1)$ Shamir sharing where the secret stored at the j-th position is $\delta_{i,j} \cdot \Delta$. At this point, the verifiers can locally convert this sharing to an additive sharing $\langle \delta_{i,j} \cdot \Delta \rangle$ by Lagrange evaluation, and further add $\langle w_{i,j} \cdot \Delta \rangle = \langle \delta_{i,j} \cdot \Delta \rangle + \langle r_{i,j} \cdot \Delta \rangle$, obtaining the desired authenticated sharing.

Difficulties in the Ring Case and our Solution. As shown in [EXY22; Abs+20; Abs+19; Bon+19], the ring $\mathbb{Z}_{p^k}$ behaves similarly to $\mathbb{F}_p$ when it comes to Shamir secret-sharing.[7] In particular, for a constant p such as $p = 2$, relevant for $\mathbb{Z}_{2^k}$, we can only use packed SS over a large enough extension ring, and at this point each secret becomes an element in such extension. If we denote by d be the extension degree, directly sharing values in the base ring using packed Shamir sharings over the extension ring would increase communication by a factor of d. To avoid such an increase in communication, we use a single extension element to "encode" d values over the base ring, by using the standard correspondence between degree-d extension elements and vectors of dimension d over

[7] These works are set in the *honest majority* regime, while our work is for dishonest majority. However, these references are relevant for us since they use Shamir secret-sharing which, as we discuss, is one of the core tools we make use of. This is enabled in the dishonest majority regime thanks to adapting the observations from SuperPack [Esc+23].

the base ring. This way, each packed Shamir sharing is effectively storing not only σ ring extension elements, but $d \cdot \sigma$ values over the base ring. This allows us to keep the same communication complexity as that in the field case.

Unfortunately, this modification renders the above way of computing the MACs for the differences $\boldsymbol{\delta}$ invalid: Recall that in the field case, we use $[\boldsymbol{\delta}_i]_{\sigma-1} \cdot [\Delta|_j]_t$ to compute the MAC for the j-th secret of $\boldsymbol{\delta}_i$, say $\delta_{i,j}$. In the ring case, $\delta_{i,j}$ is an element in the extension ring, which contains d elements $(\delta_{i,j,1}, \ldots, \delta_{i,j,d})$ over the base ring. Our goal is to compute the MAC for each $\delta_{i,j,k}$. However, since Δ is also in the extension ring, multiplying $\delta_{i,j}$ with Δ does not give us $\delta_{i,j,k} \cdot \Delta$ for all k. Our solution is to interpret Δ, which is an element over the extension ring, as a vector of d elements over the base ring, say $(\Delta_1, \ldots, \Delta_d)$. Then, we generate $[\Delta_\ell|_j]_t$ for all $\ell \in [d]$. Note that this is a (non-packed, standard) Shamir sharing over the extension ring, where the secret is Δ_ℓ, an element of the *base* ring. The main observation is that $\Delta_\ell \cdot \delta_{i,j} = \Delta_\ell \cdot (\delta_{i,j,1}, \ldots, \delta_{i,j,d}) = (\Delta_\ell \cdot \delta_{i,j,1}, \ldots, \Delta_\ell \cdot \delta_{i,j,d})$, and thus, by computing $[\boldsymbol{\delta}_i]_{\sigma-1} \cdot [\Delta_\ell|_j]_t$, we obtain packed Shamir sharings of $(\Delta_\ell \cdot \delta_{i,j,1}, \ldots, \Delta_\ell \cdot \delta_{i,j,d})$. These can be locally converted into additive sharings $\{\langle \Delta_\ell \cdot \delta_{i,j,k} \rangle\}_{k \in [d]}$. After doing this for all $\ell \in [d]$, for each $k \in [d]$, we obtain $\{\langle \Delta_\ell \cdot \delta_{i,j,k} \rangle\}_{\ell \in [d]}$. From this all parties can locally compute $\langle \Delta \cdot \delta_{i,j,k} \rangle$, as desired. Compared with the field case, we only need to generate $[\Delta_\ell|_j]_t$ for all $\ell \in [d]$. The increased amount of communication is independent of the circuit size.

Recursive Verification. Our discussion so far has focused on letting the verifiers obtain authenticated shares of the extended witness, which corresponds to steps 1-2 in our generic template from earlier in the section. What remains is step 3: verifying that the extended witness represents a valid circuit computation. Furthermore, we must do this with sublinear communication $O(n \cdot \mathsf{polylog}|C|)$. To this end we follow the approach from two previous MVZK works [Bau+22a; YW22], extending their ideas so that they work in the dishonest-majority context with potential prover-verifier collusion. The high-level idea is to use polynomials to represent all wire values, reducing the problem to checking multiplicative relations of polynomials, which can be done succinctly with the help of the Schwartz–Zippel lemma. To balance between the computation and communication overhead, these works also utilize the recursive check [Bon+19]. Our online protocol (Section 4.2) also includes a Fiat-Shamir transformation derived from [YW22], which reduces the number of rounds. Given space constraints, and since our approach is very close in techniques to these prior works, we refer the reader to Section 4.1 where the detailed protocols can be found. An technical overview can still be found in Appendix A.

1.3 Related Work

The task of multiple parties verifying the validity of a witness held by a prover has received noticeable attention in the literature. Not many works study the dishonest majority case. Here, while one approach involves leveraging maliciously secure MPC protocols within the dishonest-majority setting [Ben+11; Dam+12; GPS22; Esc+23], this method typically results in inefficient constructions with substantial round complexity. In contrast, the work presanted in [Zho+23] introduces an MVZK protocol explicitly tailored for the dishonest majority setting, but as we have mentioned it tailores the worst-case scenario of one single honest party and it is mostly of theoretical relevance.

For the honest majority case, ample literature can be found. First, as before, one can use any maliciously secure honest-majority MPC protocol [GIP15; Boy+20; GSZ20; Goy+21; GPS21; Esc+22], but this technique results in considerably elevated communication and computation expenses. In contrast, specialized MVZK protocols offer better efficiency. Applebaum, Kachlon, and Patra [AKP22] and Baum et al. [Bau+22a] each introduced MVZK protocols under the assumption that a majority of verifiers maintain honesty. In their work, Applebaum, Kachlon, and Patra [AKP22] primarily focused on a theoretical approach, emphasizing "Minicrypt"-type assumptions necessary to attain round-optimal MVZK protocols. Furthermore, the work by Abe, Cramer, and Fehr [ACF02] and Groth and Ostrovsky [GO07] presented non-interactive MVZK protocols that accommodate corruption thresholds of at most $t < n/3$ and $t < n/2$ verifiers, respectively. However, their protocols are dependent on public-key operations. Conversely, Baum et al. [Bau+22a] used a secret-sharing-based approach, aiming to develop highly efficient MVZK protocols. However, their protocols can withstand a small number of corrupted verifiers, either $t < n/3$ or $t < n/4$. The work of [YW22] achieved MVZK protocols based on secret-sharing too when the threshold of corrupted verifiers is

$t < n/2$ with communication complexity of $(1/2 + o(1))n$ field elements per multiplication gate, across all verifiers. They also proposed a solution for $t < n(1/2 - \gamma)$ with $0 < \gamma < 1/2$, and here the total communication complexity is $O(1)$ field elements per multiplication gate.

Other related works, while similar, differ from our approach. For instance, a related concept is distributed ZK as explored in [Bon+19]. This concept involves scenarios where the statement x remains unknown to any specific verifier but is instead secretly shared. However, the protocols introduced in that work only support a restricted class of languages and work in the honest majority case or does not allow collusion between the prover and the verfiers. The work of [Boy+20] can be viewed as an MVZK protocol in the honest majority case but it has worst efficiency than [YW22].

Finally, we point out that ZK with multiple verifiers has not been explored over more general rings than finite fields, despite notable works such as [LXY23; Bau+22b] considering such rings in the standard two-party setting where there is one verifier and one prover.

2 Preliminaries

Basic Settings. We consider a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$, among which $t = n(1-\epsilon)$ can be actively corrupted, where $\epsilon \in (0,1)$. Let $\mathcal{R}$ be a ring of the form $\mathbb{Z}/p^k\mathbb{Z}$ for some prime p and an integer $k \geq 1$. We consider the task of verifying circuits $\mathcal{C} : \mathcal{R}^{|\mathcal{I}|} \to \{0,1\}$ over $\mathcal{R}$. Formally, this is captured by a functionality, which we denote by $\mathcal{F}_{\mathsf{ZK}}$, which is defined as follows: take input $\boldsymbol{\omega} \in \mathcal{R}^{|\mathcal{I}|}$ from $\mathcal{P}$, if $\mathcal{C}(\boldsymbol{\omega}) = 0$, return accept to $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, and otherwise, return reject. In this work, we only assume that there is a secure point-to-point communication channel between every pair of parties (including both the prover and verifiers). In particular, we do not require the existence of a reliable broadcast channel among the parties.

Galois Rings. For the whole paper we let $\mathcal{K} = \mathsf{GR}(p^k, d)$ be the degree-d Galois ring extension of $\mathbb{Z}/p^k\mathbb{Z}$ such that $p^d \geq 2^\kappa$, where κ is the security parameter. Formally, $\mathcal{K}$ is the quotient ring $\mathcal{R}[\mathtt{X}]/(f(\mathtt{X}))$ where $f(\mathtt{X})$ is a monic polynomial over $\mathcal{R}$ that is irreducible over $\mathbb{F}_p$ when taken modulo p. For $k = 1$, Galois rings are simply field extensions. In fact, a crucial fact we will use throughout our work is that $\mathsf{GR}(p^k, d)$ is, in a way, *equivalent* to $\mathbb{F}_{p^d}$: it is possible to perform unique polynomial interpolation over $\mathsf{GR}(p^k, d)$ in the same way as over $\mathbb{F}_{p^d}$, which in turns enables essential techniques such Schwartz-Zippel lemma, widely used for compressing checks. We refer the reader to several works that have used Galois rings for more details on these structures [EXY22; Abs+20; Abs+19; Bon+19].

Secret Sharing Schemes. Let $\sigma = n - t = n\epsilon$ be the number of honest parties. Taking $\mathcal{K} = \mathsf{GR}(p^k, d)$ with $p^d > n + \sigma$ (which is the case since $p^d > 2^\kappa \gg n + \sigma$), there exists a subset $\{\beta_1, \ldots, \beta_\sigma, \alpha_1, \ldots \alpha_n\}$ where all pairwise non-zero differences are invertible in $\mathcal{K}$, and this enables packed Shamir secret sharings with α's as the "shares evaluation points" and β's as the "secret-evaluation points" [Abs+19]. More precisely, for $\boldsymbol{x} \in \mathcal{K}^\sigma$ we use the notation $[\boldsymbol{x}]_\delta$ when each verifier $\mathcal{V}_i$ has $f(\alpha_i)$ for some polynomial f over $\mathcal{K}$ of degree at most δ such that $\boldsymbol{x} = (f(\beta_1), \ldots, f(\beta_\sigma))$. Any secret $\boldsymbol{x} \in \mathcal{K}^\sigma$ can be shared with a degree $t + (\sigma - 1) = n - 1$ polynomial without leaking anything to the adversary, and any public vector $\boldsymbol{c} \in \mathcal{K}^\sigma$ can be "shared" with a degree $\sigma - 1$ polynomial, which we exploit in our protocols to be able to multiply public values by secret-shared vectors. Finally, $[\cdot]$ is standard packed secret-sharing [FY92], except that it is described over Galois rings, and so all of the usual properties hold: local addition and local multiplication while summing up the degrees.

For $x \in \mathcal{K}$, we use $\langle x \rangle$ to denote an additive secret-sharing of x among the n verifiers, and we denote $[\![x]\!] = (\langle x \rangle, \langle \Delta \cdot x \rangle, \langle \Delta \rangle)$, where $\Delta \in \mathcal{K}$ is a random global key. Sometimes we use this for $x \in \mathcal{R}$, which is covered by the notation since $\mathcal{R} \subseteq \mathcal{K}$. For a shared vector $[\boldsymbol{x}]_{n-1}$ with $\boldsymbol{x} \in \mathcal{K}^\sigma$, the parties can locally obtain *additive* shares of each entry $\langle x_i \rangle$, where $x_i \in \mathcal{K}$. Since $\mathcal{K} \cong \mathcal{R}^d$, this can in turn be locally converted to additive shares of each $\langle x_{ij} \rangle$, where x_i is interpreted as $(x_{i1}, \ldots, x_{id}) \in \mathcal{R}^d$.

Assumed Functionalities. We assume some standard functionalities. $\mathcal{F}_{\mathsf{Coin}}$ enables the verifiers to sample public random coins, and $\mathcal{F}_{\mathsf{Commit}}$ allows a verifier to commit to some input, revealing it at a later stage.

3 Preprocessing Phase

In this section, we introduce the preprocessing of our protocol. We consider the verification circuit over a ring $\mathcal{R}$ of the form of $\mathbb{Z}/p^k\mathbb{Z}$ where p is a prime and $k \geq 1$. We let $\mathcal{K} = \mathsf{GR}(p^k, d)$ be the degree-d Galois ring extension of $\mathbb{Z}/p^k\mathbb{Z}$ such that $p^d \geq 2^\kappa$, where κ is the security parameter.

3.1 Programmable Vector OLE

In this work we make use of a programmable vector oblivious linear evaluation (VOLE) functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, which involves only two parties P_A and P_B. For an output length m, it allows P_A with input $\boldsymbol{u} \in \mathcal{R}^m$ and P_B with input $\Delta \in \mathcal{K}$ to compute additive shares of $\boldsymbol{u} \cdot \Delta$. At the end of the protocol, P_A and P_B hold $\boldsymbol{w}$ and $\boldsymbol{v} \in \mathcal{K}^m$, respectively, such that $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$. Compared to an ordinary VOLE protocol, the programmability aspect refers to the fact that the functionality allows P_A to choose their input $\boldsymbol{u}$ by providing a short seed $\mathsf{seed} \in S$, which is expanded with a function $\mathsf{Expand} : S \to \mathcal{R}^m$. S is defined as the seed space and depends on the instantiation of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. We borrow the functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ from Le Mans [RS22], which is defined specifically over fields, and extend it to the general ring case. We provide an instantiation of this functionality in Appendix B.

Functionality 1: $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$

Parameters: A seed space S. An expansion function $\mathsf{Expand} : S \to \mathcal{R}^m$. Two parties P_A and P_B.
Initialize: Upon receiving init from P_A and (init, Δ) from P_B, store Δ and ignore all subsequent init commands.
Extend: Upon receiving $(\mathsf{extend}, \mathsf{seed})$ from P_A in which $\mathsf{seed} \in S$, and extend from P_B, do:
1. Compute $\boldsymbol{u} \leftarrow \mathsf{Expand}(\mathsf{seed})$.
2. Sample uniform $\boldsymbol{v} \leftarrow \mathcal{K}^m$ and compute $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$.
   - If P_B is corrupted, receive $\boldsymbol{v}$ from $\mathcal{A}$ and compute $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$.
   - If P_A is corrupted, receive $\boldsymbol{w}$ from $\mathcal{A}$ and compute $\boldsymbol{v} = \boldsymbol{w} - \boldsymbol{u} \cdot \Delta$.
3. If P_B is corrupted, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to $\mathcal{A}$ and continue. Else, send abort to both parties, output seed to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

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

3.2 Multi-Party Vector Oblivious Linear Evaluation

In Le Mans [RS22], the functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ is used to instantiate an n-party VOLE functionality that allows all parties to efficiently generate authenticated random additive sharings. At a high level, the n-party VOLE functionality generates programmed random VOLE pairs (as described in $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$) for every pair of parties such that the i-th party uses the same seed^i and Δ^i in all VOLE pairs.

We modify the functionality in Le Mans [RS22] to allow a designated party $\mathcal{P}$ (which will be used as the prover in our case) to choose the expansion function from a predefined class of expansion functions and sample the seeds for all verifiers. In this way, $\mathcal{P}$ can obtain all shares by expanding the seeds locally. We require that a randomly sampled expansion function from the predefined class of expansion functions is a PRG. We describe our n-party VOLE functionality $\mathcal{F}_{\mathsf{nVOLE}}$ as follows. To instantiate $\mathcal{F}_{\mathsf{nVOLE}}$, we modify the protocol in [RS22] to let $\mathcal{P}$ sample the seeds and distribute them to all verifiers.

Functionality 2: $\mathcal{F}_{\mathsf{nVOLE}}$

Parameters: A seed space S. A class of expansion functions defined over $S \to \mathcal{R}^m$. $n+1$ parties $\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n$.
Initialize:
1. For $i \in [n]$, upon receiving init from $\mathcal{V}_i$, sample uniform $\Delta^i \leftarrow \mathcal{K}$ and send it to $\mathcal{V}_i$. If $\mathcal{V}_i$ is corrupted, receive $\Delta^i \in \mathcal{K}$ from $\mathcal{V}_i$. Ignore all subsequent init commands from $\mathcal{V}_i$.

2. If $\mathcal{P}$ is corrupted, receive Expand from the class of expansion functions from $\mathcal{P}$. Otherwise, sample a random function as Expand from the class of expansion functions. Then, send Expand to all parties.

Extend: This procedure can be repeated multiple times. Receive (extend, m) from all parties and do:

1. Depending on whether $\mathcal{P}$ is corrupted, do the following.
   - If $\mathcal{P}$ is corrupted, receive seed^i from $\mathcal{P}$ and set $\overline{\mathsf{seed}}^i = \mathsf{seed}^i$ for all $i \in [n]$.
   - Otherwise, uniformly sample $\mathsf{seed}^i \leftarrow S$ for all $i \in [n]$. For each corrupted verifier $\mathcal{V}_i$, send seed^i to $\mathcal{A}$ and receive $\overline{\mathsf{seed}}^i$. For each honest verifier $\mathcal{V}_i$, set $\overline{\mathsf{seed}}^i = \mathsf{seed}^i$.

   Compute $\boldsymbol{u}^i \leftarrow \mathsf{Expand}(\overline{\mathsf{seed}}^i)$ for all $i \in [n]$.
2. For all i and $j \in [n]\backslash\{i\}$, sample uniform $\boldsymbol{v}_i^j \leftarrow \mathcal{K}^m$ and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{v}_i^j$.
   - If $\mathcal{V}_j$ is corrupted, receive $\boldsymbol{v}_i^j$ from $\mathcal{V}_j$, and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{v}_i^j$.
   - Otherwise, if $\mathcal{V}_i$ is corrupted, receive $\boldsymbol{w}_j^i$ from $\mathcal{V}_i$ and compute $\boldsymbol{v}_i^j = \boldsymbol{w}_j^i - \boldsymbol{u}^i \cdot \Delta^j$.
3. For all $i \in [n]$, output seed^i to $\mathcal{P}$ and $(\overline{\mathsf{seed}}^i, \boldsymbol{u}^i, \{\boldsymbol{w}_j^i\}_{j\neq i})$ to $\mathcal{V}_i$.
4. For all $i \in [n]$ and $j \in [n]\backslash\{i\}$, output $\boldsymbol{v}_i^j$ to $\mathcal{V}_j$.

Protocol 1: Π_{nVOLE}

Parameters: Ring $\mathcal{R}$, degree-d Galois ring $\mathcal{K}$, a seed space S, a class of expansion functions defined over $S \to \mathcal{R}^m$, and parties $\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n$.

Initialize:

1. Each verifier $\mathcal{V}_i$ uniformly samples $\Delta^i \in \mathcal{K}$. For each pair (i, j) such that $i \neq j$ and $i, j \in [n]$, $\mathcal{V}_i$ sends init and $\mathcal{V}_j$ sends $(\mathsf{init}, \Delta^j)$ to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$.
2. $\mathcal{P}$ uniformly samples Expand from the predefined set of expansion functions and sends Expand to $\mathcal{V}_1, \ldots, \mathcal{V}_n$. Then each of $\mathcal{V}_1, \ldots, \mathcal{V}_n$ computes $\mathsf{H}(\mathsf{Expand})$ and sends it to each other to check the consistency of Expand. If a verifier receives different hash values from other verifiers, this verifier aborts.

Extend: This procedure can be repeated multiple times. On an agreed output length m, do:

1. For $i \in [n]$, $\mathcal{P}$ samples a uniform seed seed^i and sends it to $\mathcal{V}_i$.
2. For each pair (i, j) such that $i \neq j$ and $i, j \in [n]$, $\mathcal{V}_i$ sends $(\mathsf{extend}, \mathsf{seed}^i)$ and $\mathcal{V}_j$ sends extend to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. For $\ell \in [m+1]$, $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ returns $(u_\ell^i, w_{j,\ell}^i)$ to $\mathcal{V}_i$ and $v_{i,\ell}^j$ to $\mathcal{V}_j$ where $u_{m+1}^i \in \mathcal{K}$. Here we take the output length of Expand in $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ to be $m+d$ and view the last d outputs in $\mathcal{R}$ as an element in $\mathcal{K}$. $\mathcal{V}_i$ and $\mathcal{V}_j$ can locally compute $w_{j,m+1}^i, v_{i,m+1}^j$ respectively such that $w_{j,m+1}^i = u_{m+1}^i \cdot \Delta^j + v_{i,m+1}^j$.

Consistency Check: Verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ check the correctness of generated authenticated values.

1. Send $(\mathsf{rand}, \mathcal{K})$ to $\mathcal{F}_{\mathsf{Coin}}$, which returns $\chi \in \mathcal{K}$.
2. Each verifier $\mathcal{V}_i$ computes $u^i = \sum_{\ell=1}^m \chi^\ell \cdot u_\ell^i + u_{m+1}^i$ and defines $\langle u \rangle = (u^1, \ldots, u^n)$.
3. For $i \in [n]$, $\mathcal{V}_i$ randomly samples $\langle b_i \rangle = (b_i^1, \ldots, b_i^n)$ such that $b_i = \sum_{j=1}^n b_i^j = 0$. It distributes shares to other verifiers.
4. All verifiers locally compute $\langle \hat{u} \rangle = \langle u \rangle + \sum_{i=1}^n \langle b_i \rangle$ and send their shares to all verifiers. Then all verifiers compute the secret $\hat{u}$ from other verifiers' shares.
5. For $i \in [n]$, $\mathcal{V}_i$ invokes $\mathcal{F}_{\mathsf{Commit}}$ with inputs

$$u^i, \{Z_j^i\}_{j\neq i} = \{w_j^i\}_{j\neq i}, Z_i^i = (u^i - \hat{u}) \cdot \Delta^i - \sum_{j\neq i} v_j^i,$$

   where $w_j^i = \sum_{\ell=1}^m \chi_\ell \cdot w_{j,\ell}^i + w_{j,m+1}^i$ and $v_j^i = \sum_{\ell=1}^m \chi_\ell \cdot v_{j,\ell}^i + v_{j,m+1}^i$.
6. After receiving the commitments from all other verifiers, all verifiers check the consistency of all received messages (including $\langle \hat{u} \rangle$ and all commitments). This is done by letting each verifier compute a hash of his received messages and send the hash result to all verifiers. If a verifier receives different hash values from other verifiers, this verifier aborts.
7. All verifiers open their commitments of $u^i, \{Z_j^i\}_{j=1}^n$ and each verifier $\mathcal{V}_k$ checks the following:
   - $\hat{u} \stackrel{?}{=} \sum_{i=1}^n u^i$
   - For all $j \in [n]\backslash\{k\}$, $Z_k^j \stackrel{?}{=} v_j^k + u^j \cdot \Delta^k$.
   - For all $j \in [n]$, $\sum_{i=1}^n Z_j^i \stackrel{?}{=} 0$.

   If any check fails, $\mathcal{V}_k$ aborts.

Output: The output of each party is specified as follows.

- $\mathcal{P}$ outputs seed^i for all $i \in [n]$.
- Each $\mathcal{V}_i$ outputs seed^i, $\{u_\ell^i\}_{\ell\in[m]}$, $\{w_{j,\ell}^i\}_{\ell\in[m],j\neq i}$, and $\{v_{j,\ell}^i\}_{\ell\in[m],j\neq i}$.

We show that Π_{nVOLE} securely computes $\mathcal{F}_{\mathsf{nVOLE}}$ against the following two types of adversaries:

- The first type corrupts up to t verifiers among $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, but not $\mathcal{P}$. This corresponds to the case where the prover is honest and up to t verifiers collude.
- The second type corrupts up to t verifiers among $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ together with $\mathcal{P}$. This corresponds to the case where the prover is corrupted and colludes with up to t verifiers.

Theorem 1. *Assume that a random function from the class of expansion functions in Π_{nVOLE} and $\mathcal{F}_{\mathsf{nVOLE}}$ is a PRG. Protocol Π_{nVOLE} securely realizes $\mathcal{F}_{\mathsf{nVOLE}}$ in the presence of a malicious adversary $\mathcal{A}$ who can corrupt $\mathcal{P}$ and up to t-out-of-n verifiers among $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ in the $(\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid and random oracle model.*

We refer the readers to Section C.1 for the proof.

Remark 1. In Π_{nVOLE}, we ask the prover $\mathcal{P}$ to send the description of Expand to all verifiers. This is to avoid the assumption of CRS for the parameters in the LPN assumption we use over rings. To be more concrete, the expansion function is derived from the parameters of the LPN assumption. And the expansion function is a pseudorandom generator only if the parameters of the LPN assumption are randomly sampled. We note that the expansion function is used to generate random authenticated additive sharings that are used to protect the secrecy of $\mathcal{P}$. Thus it is sufficient to ask $\mathcal{P}$ to sample the expansion function. We also let $\mathcal{P}$ choose the seeds for all verifiers. This allows $\mathcal{P}$ to start computing the differences that are shared to the verifiers in the online phase. Note that when $\mathcal{P}$ is corrupted, the expansion function may not be a PRG and the seeds may not be chosen from random. This is fine since we do not need to protect the secrecy of $\mathcal{P}$ when he is corrupted. In the security proof of our construction, we only assume that the expansion function is a PRG and the seeds are uniformly sampled when $\mathcal{P}$ is honest.

Another instantiation (which is used in our experiment) is to assume that all parties start with uniformly random parameters for the LPN assumption. In this case, all parties automatically agree on the expansion function and it is a PRG due to the LPN assumption. And with this instantiation, one optimization is to let each verifier $\mathcal{V}_i$ generate his own seed seed^i rather than receiving it from $\mathcal{P}$. Then all verifiers send their seeds at the end of Π_{nVOLE} (which can be done in parallel with Step (5) in Π_{Prep}). This allows us to remove the first round communication from $\mathcal{P}$ to all verifiers.

3.3 Preprocessing Protocol

In the preprocessing, all parties prepare random authenticated additive sharings. This is done by invoking $\mathcal{F}_{\mathsf{nVOLE}}$. Recall that in $\mathcal{F}_{\mathsf{nVOLE}}$,

- Each verifier $\mathcal{V}_i$ holds Δ^i. We set $\Delta = \sum_{i=1}^n \Delta^i$
- For all $\ell \in [m]$, for every pair of verifiers $(\mathcal{V}_i, \mathcal{V}_j)$, $\mathcal{V}_i$ holds $u_\ell^i, w_{j,\ell}^i$ and $\mathcal{V}_j$ holds $v_{i,\ell}^j$ such that $w_{j,\ell}^i = u_\ell^i \cdot \Delta^j + v_{i,\ell}^j$.

Recall that there are at most t corrupted verifiers. Following the observation in [Esc+23], all verifiers can view $(u_\ell^1, \ldots, u_\ell^n)$ as a degree-$(n-1)$ packed Shamir sharing that stores $\sigma = n - t$ random values. Let $\boldsymbol{r}_\ell = (r_{\ell,1}, \ldots, r_{\ell,\sigma})$ denote the secret vector of this packed Shamir sharing. All verifiers can locally convert it to σ additive sharings $\langle r_{\ell,1}\rangle, \ldots, \langle r_{\ell,\sigma}\rangle$. In particular, by using $w_{j,\ell}^i, v_{i,\ell}^j$, all verifiers can locally compute $\langle \Delta \cdot r_{\ell,1}\rangle, \ldots, \langle \Delta \cdot r_{\ell,\sigma}\rangle$. In this way, we obtain σ authenticated random sharings $[\![r_{\ell,1}]\!], \ldots, [\![r_{\ell,\sigma}]\!]$, where $[\![r_{\ell,i}]\!] = (\langle r_{\ell,i}\rangle, \langle \Delta \cdot r_{\ell,i}\rangle)$. In total, we obtain $m \cdot \sigma$ random authenticated random sharings. Note that the prover $\mathcal{P}$ can compute the secret vector $\boldsymbol{r}_\ell$ for all ℓ as well. Jumping ahead, with these random authenticated additive sharings, $\mathcal{P}$ will distribute the actual wire values by distributing the differences between the wire values and the secrets of these random authenticated additive sharings.

We also let each verifier $\mathcal{V}_i$ share his authentication key Δ^i. We naturally view $\Delta^i \in \mathcal{K}$ as a vector of d elements $(\Delta_1^i, \ldots, \Delta_d^i)$ in $\mathcal{R}$. For each $j \in [d]$ and $\ell \in [\sigma]$, we ask $\mathcal{V}_i$ to share Δ_j^i using a Shamir secret sharing over $\mathcal{K}$ such that the secret is stored at position ℓ, denoted by $[\Delta_j^i|_\ell]_t$. These degree-t Shamir sharings are used to compute the MACs for the differences distributed by the prover in the online phase. We summarize the preprocessing protocol as follows.

Protocol 2: Π_{Prep}

Define an arithmetic circuit $\mathcal{C}$ with $|\mathcal{I}|$ input wires and $|\mathcal{C}|$ multiplication gates. Let c be the number of random authenticated additive sharings needed during the online consistency check. Define $M = \lceil (|\mathcal{I}| + |\mathcal{C}| + c \cdot d)/\sigma \rceil$, where recall that d is the extension degree of $\mathcal{K}$ with respect to $\mathcal{R}$ and $\sigma = n - t$ is the number of honest verifiers.

1. For $i \in [n]$, the verifier $\mathcal{V}_i$ sends init to $\mathcal{F}_{\mathsf{nVOLE}}$, which returns $\Delta^i \in \mathcal{K}$ to $\mathcal{V}_i$. Define $\Delta = \sum_{i\in[n]} \Delta^i$. $\mathcal{V}_i$ also receives the expansion function Expand from $\mathcal{F}_{\mathsf{nVOLE}}$.
2. $\mathcal{P}$ sends (extend, M) to $\mathcal{F}_{\mathsf{nVOLE}}$ and receives $\{\mathsf{seed}^i\}_{i=1}^n$. For $i \in [n]$, $\mathcal{V}_i$ sends (extend, M) to $\mathcal{F}_{\mathsf{nVOLE}}$, which returns $\overline{\mathsf{seed}}^i$, $\boldsymbol{u}^i$, $\{\boldsymbol{w}_j^i, \boldsymbol{v}_j^i\}_{j\neq i}$ to $\mathcal{V}_i$. For all $\mathcal{V}_i, \mathcal{V}_j$, $\mathcal{V}_i$ holds $\boldsymbol{u}^i, \boldsymbol{w}_j^i$ and $\mathcal{V}_j$ holds $\Delta^j, \boldsymbol{v}_i^j$ such that $\boldsymbol{v}_i^j = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{w}_j^i$.
3. For all $\ell \in [M]$, all verifiers run the following steps to obtain random authenticated additive sharings.
   (a) All verifiers view $(u_\ell^1, u_\ell^2, \ldots, u_\ell^n)$ as a degree-$(n-1)$ packed Shamir sharing. Let $\boldsymbol{r}_\ell = (r_{\ell,1}, \ldots, r_{\ell,\sigma})$ be the secret vector of this packed Shamir sharing. Specifically, these exists a degree-$(n-1)$ polynomial $f(\cdot)$ that satisfies $f(\alpha_i) = u_\ell^i$ for $i \in [n]$ and $f(\beta_j) = r_{\ell,j}$ for $j \in [\sigma]$, where $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$ form an exceptional sequence.
   (b) Verifiers locally convert $[\boldsymbol{r}_\ell]_{n-1}$ to an additive sharing $\langle r_{\ell,j}\rangle$ via polynomial evaluation to β_j-th coordinate for $j \in [\sigma]$. Concretely, denote i-th share of $[\boldsymbol{r}_\ell]_{n-1}$ to be $[\boldsymbol{r}_\ell]_{n-1}^i$ and $L_i(\cdot)$ to be the i-th Lagrange polynomial. We have $\langle r_{\ell,j}\rangle = (L_i(\beta_j) \cdot [\boldsymbol{r}_\ell]_{n-1}^i)_{i=1}^n$.
   (c) Each verifier $\mathcal{V}_{i_1}$ computes his share of $\langle \Delta \cdot r_{\ell,j}\rangle$ by $L_{i_1}(\beta_j) \cdot u_\ell^{i_1} \cdot \Delta^{i_1} + \sum_{i_2\neq i_1} (L_{i_2}(\beta_j) \cdot v_{i_2,\ell}^{i_1} - L_{i_1}(\beta_j) \cdot w_{i_2,\ell}^{i_1})$. The correctness follows from the fact that

$$
\begin{aligned}
&\sum_{i_1=1}^{n} \sum_{i_2\neq i_1} (L_{i_2}(\beta_j) \cdot v_{i_2,\ell}^{i_1} - L_{i_1}(\beta_j) \cdot w_{i_2,\ell}^{i_1}) \\
=&\sum_{i_1=1}^{n} \sum_{i_2\neq i_1} (L_{i_1}(\beta_j) \cdot v_{i_1,\ell}^{i_2} - L_{i_1}(\beta_j) \cdot w_{i_2,\ell}^{i_1}) \\
=&\sum_{i_1=1}^{n} \sum_{i_2\neq i_1} L_{i_1}(\beta_j) \cdot u_\ell^{i_1} \cdot \Delta^{i_2} \\
=&\sum_{i_1=1}^{n} L_{i_1}(\beta_j) \cdot u_\ell^{i_1} \cdot (\Delta - \Delta^{i_1}) \\
=&\Delta \cdot r_{\ell,j} - \sum_{i_1=1}^{n} L_{i_1}(\beta_j) \cdot u_\ell^{i_1} \cdot \Delta^{i_1}.
\end{aligned}
$$

   (d) Since $\{\boldsymbol{u}^i\}_{i=1}^n$ are fully determined by $\{\mathsf{seed}^i\}_{i=1}^n$, $\mathcal{P}$ follows the similar steps to compute $r_{\ell,j}$ for all $\ell \in [M]$ and $j \in [\sigma]$.
   (e) All verifiers convert the last $c \cdot d$ authenticated additive sharings in $\mathcal{R}$ to c authenticated additive sharings in $\mathcal{K}$.
4. Each verifier $\mathcal{V}_i$ views $\Delta^i \in \mathcal{K}$ as a vector of d elements in $\mathcal{R}$. For $i \in [n], j \in [d], \ell \in [\sigma]$, $\mathcal{V}_i$ constructs and distributes a degree-t Shamir secret sharing $[\Delta_j^i|_\ell]_t$. Verifiers compute $[\Delta_j|_\ell]_t = \sum_{i=1}^n [\Delta_j^i|_\ell]_t$.
5. Each verifier $\mathcal{V}_i$ samples a random κ-bit string nonce_i and sends it to the prover $\mathcal{P}$.

Remark 2. We note that the random authenticated additive sharings are obtained by performing local computation on the data verifiers received from $\mathcal{F}_{\mathsf{nVOLE}}$. In $\mathcal{F}_{\mathsf{nVOLE}}$, when $\mathcal{P}$ is honest, corrupted verifiers may use different seeds from those generated by $\mathcal{P}$. Note that corrupted verifiers also receive the correct seeds from $\mathcal{P}$, which means that the adversary can also compute the correct shares corrupted verifiers should hold. Eventually, this will translate to additive attacks to the wire values shared by $\mathcal{P}$ in the online phase, which will be caught by the online verification.

Remark 3. In the preprocessing phase, we do not check the degree-t Shamir sharings of the MAC keys shared by all verifiers. It means that corrupted verifiers can share different MAC keys from those sent to $\mathcal{F}_{\mathsf{nVOLE}}$. This is OK because these sharings are only used to compute additive sharings of the MAC values of the differences (between the actual wire values and the secrets of random authenticated additive sharings) shared by $\mathcal{P}$ in the online phase. Since the differences are public

values, corrupted verifiers can still compute the correct shares of the additive sharings of the MAC values. Thus, even if corrupted verifiers share incorrect MAC keys, we can always assume that they eventually hold correct shares (although they may choose to not use correct shares in the computation).

Remark 4. In the preprocessing, we ask each verifier to provide a nonce to the prover. These nonces are used to bind the shares chosen by the prover. In the online phase, the prover will generate random challenges via Fiat-Shamir by taking all nonces as input.

4 Online Phase

Define an arithmetic circuit $\mathcal{C}$ with $|\mathcal{I}|$ input wires and $|\mathcal{C}|$ multiplication gates. Let c be the number of random authenticated additive sharing needed during the consistency check (which will be specified later). Let $N = |\mathcal{I}| + |\mathcal{C}|$. Recall that in the preprocessing phase, all verifiers have prepared random authenticated additive sharings $\{[\![r_i]\!]\}_{i=1}^{N}$ in $\mathcal{R}$ together with c random authenticated additive sharings in $\mathcal{K}$. In particular, the shares are generated from the seeds sampled by $\mathcal{P}$. Thus, $\mathcal{P}$ can compute the secrets of all sharings.

In the online phase, the prover $\mathcal{P}$ will first share the actual wire values to all verifiers. $\mathcal{P}$ first computes all wire values in $\mathcal{C}$. Then for each input wire of $\mathcal{C}$ and the output wire of each multiplication gate, we assign one random authenticated random sharing. For each of these wires, $\mathcal{P}$ computes the difference λ between the actual wire value v and the secret r of the random additive sharing. By distributing all differences, all verifiers can obtain an authenticated additive sharing for each wire value. However, doing this naively would cost $O(|\mathcal{C}| \cdot n)$ communication. Our idea is to use degree-$(\sigma - 1)$ packed Shamir sharings in $\mathcal{K}$ to share those differences. Here we choose the degree to be $\sigma - 1$ so that we can use the degree-t Shamir sharing of the MAC keys prepared in the preprocessing phase to let all verifiers locally compute additive sharings of the MAC values of these differences. Also, a degree-$(\sigma - 1)$ packed Shamir sharing is fully decided by the shares of honest verifiers, where recall that $\sigma = n - t$ is the number of honest verifiers. To be more concrete, for each group of $\sigma \cdot d$ values $\boldsymbol{D} = \{\lambda_{i,j}\}_{i\in[\sigma],j\in[d]}$ in $\mathcal{R}$, $\mathcal{P}$ views them as σ elements $\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_\sigma$ in $\mathcal{K}$ and then computes the degree-$(\sigma - 1)$ packed Shamir sharing $[\boldsymbol{D}]_{\sigma-1}$ determined by those σ elements. Finally $\mathcal{P}$ distributes the shares of $[\boldsymbol{D}]_{\sigma-1}$ to all verifiers.

To compute the MAC values of the differences shared by $\mathcal{P}$, observe that by computing $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell|_i]_t$, we obtain a degree-$(n-1)$ Shamir sharing of $(\lambda_{i,1}, \ldots, \lambda_{i,d}) \cdot \Delta_\ell$ (Recall that $\Delta_\ell \in \mathcal{R}$). Then all verifiers can locally compute an additive sharing of $\lambda_{i,j} \cdot \Delta_\ell$. By dong this for all ℓ, all verifiers can locally compute an additive sharing of $\lambda_{i,j} \cdot \Delta$. Thus eventually, all verifiers can compute authenticated additive sharings of all wire values in $\mathcal{C}$.

To check the correctness of the computation, we adapt the techniques in [Bon+19; GSZ20] from the MPC setting with honest majority over fields to our setting and extend it to the Galois ring case. We make it secure against dishonest-majority verifiers by equipping the shared elements with information-theoretic MACs. We give the protocol description in the next subsection for the check of multiplication relations.

4.1 Inner Product Check

In this subsection, we describe the protocols for checking an inner-product tuple of dimension N. In the beginning, all verifiers hold an inner-product tuple where each value is shared by an authenticated additive sharing, denoted by $(\{([\![x_i]\!], [\![y_i]\!])\}_{i=1}^{N}, [\![z]\!])$. The goal is to check whether $z = \sum_{i=1}^{N} x_i \cdot y_i$. The idea in [Bon+19; GSZ20] is to recursively reduce the dimension of the inner-product tuple until it becomes a multiplication tuple. The dimension-reduction step is done with the help from the prover and appears in $\pi_{\mathsf{VerifyIPProc}}$. In the last round of dimension reduction, the prover will share a random multiplication triple to ease the check of the final triple, which is described in $\pi_{\mathsf{VerifyIPFinal}}$.

Procedure 1: $\pi_{\mathsf{VerifyIPProc}}$

Parameters: Input dimension $m\ell$, compression parameter m and output dimension ℓ. Let $\mathsf{H} : \{0,1\}^* \to \mathcal{K}$ be a random oracle.
Input: An authenticated inner-product triple $(\{([\![x_i]\!], [\![y_i]\!])\}_{i=1}^{m\ell}, [\![z]\!])$ such that $z = \sum_{i=1}^{m\ell} x_i \cdot y_i$.
Output: An authenticated inner-product triple $(\{([\![a_i]\!], [\![b_i]\!])\}_{i=1}^{\ell}, [\![c]\!])$ such that $c = \sum_{i=1}^{\ell} a_i \cdot b_i$.
Protocol:

1. Split $\{[\![x_i]\!]\}_{i=1}^{m\ell}$ and $\{[\![y_i]\!]\}_{i=1}^{m\ell}$ into m sub-vectors each of dimension ℓ, and denote them as $\{([\![\boldsymbol{x}_j]\!], [\![\boldsymbol{y}_j]\!])\}_{j=1}^{m}$, in which $\boldsymbol{x}_j, \boldsymbol{y}_j$ are of dimension ℓ.
2. $\mathcal{P}$ fetches the next unused $m-1$ random authenticated additive sharings in $\mathcal{K}$ prepared in the preprocessing phase and denotes them as $([\![r_1]\!], \ldots, [\![r_{m-1}]\!])$. For $j \in [m-1]$, $\mathcal{P}$ computes $z_j = \sum_{i=1}^{\ell} \boldsymbol{x}_j[i] \cdot \boldsymbol{y}_j[i]$ and sends $b_j = z_j - r_j$ to verifiers. Verifiers compute $[\![z_j]\!] = b_j + [\![r_j]\!]$.
3. Parties compute $[\![z_m]\!] = [\![z]\!] - \sum_{j=1}^{m-1} [\![z_j]\!]$.
4. Compression phase.
   (a) Interpolate degree-$(m-1)$ polynomials
   $$([\![f_1(\cdot)]\!], \ldots, [\![f_\ell(\cdot)]\!]), ([\![g_1(\cdot)]\!], \ldots, [\![g_\ell(\cdot)]\!])$$
   such that $f_i(j) = \boldsymbol{x}_j[i], g_i(j) = \boldsymbol{y}_j[i]$ for $i \in [\ell], j \in [m]$.
   (b) $\mathcal{P}$ fetches the next unused $m-1$ authenticated values and denotes them as $(\tilde{r}_1, \ldots, \tilde{r}_{m-1})$. For $j \in [m-1]$, $\mathcal{P}$ computes $d_j = \sum_{i=1}^{\ell} f_i(m+j) \cdot g_i(m+j)$ and sends $c_j = d_j - \tilde{r}_j$ to verifiers. Verifiers compute $[\![d_j]\!] = c_j + [\![\tilde{r}_j]\!]$.
   (c) Parties interpolate the degree-$2(m-1)$ polynomial $[\![h(\cdot)]\!]$ such that $h(j) = z_j$ for $j \in [m]$ and $h(j) = d_{j-m}$ for $j \in [m+1, 2m-1]$.
5. Parties compute
   $$\eta \leftarrow \mathsf{H}(\mathsf{IP}, m\ell, \eta_{\mathsf{prev}}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m-1}),$$
   in which the first two parameters are used as the identifier and η_{prev} is the hash result from the previous invocation of H. If $\eta \in [m]$, all parties abort. Parties output $(\{([\![a_i]\!], [\![b_i]\!])\}_{i=1}^{\ell}, [\![c]\!])$ such that $a_i = f_i(\eta)$, $b_i = g_i(\eta)$ and $c = h(\eta)$.

Procedure 2: $\pi_{\mathsf{VerifyIPFinal}}$

Parameters: Input dimension m. Let $\mathsf{H} : \{0,1\}^* \to \mathcal{K}$ be a random oracle.
Input: An authenticated inner product triple $(\{([\![x_i]\!], [\![y_i]\!])\}_{i=1}^{m}, [\![z]\!])$ such that $z = \sum_{i=1}^{m} x_i \cdot y_i$.
Output: An authenticated multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ such that $c = a \cdot b$.
Protocol:

1. $\mathcal{P}$ uniformly samples a triple $(x_{m+1}, y_{m+1}, z_{m+1})$ such that $z_{m+1} = x_{m+1} \cdot y_{m+1}$. Fetch the next 3 unused authenticated values and denote them as $([\![\hat{r}_1]\!], [\![\hat{r}_2]\!], [\![\hat{r}_3]\!])$. $\mathcal{P}$ sends $(\hat{b}_1, \hat{b}_2, \hat{b}_3) := (x_{m+1} - \hat{r}_1, y_{m+1} - \hat{r}_2, z_{m+1} - \hat{r}_3)$ to all verifiers. Verifiers compute $[\![x_{m+1}]\!] = \hat{b}_1 + [\![\hat{r}_1]\!]$, and $[\![y_{m+1}]\!] = \hat{b}_2 + [\![\hat{r}_2]\!]$ and $[\![z_{m+1}]\!] = \hat{b}_3 + [\![\hat{r}_3]\!]$.
2. Interpolate degree-m polynomials $([\![f(\cdot)]\!], [\![g(\cdot)]\!])$ such that $f(j) = x_j, g(j) = y_j$ for $j \in [m+1]$.
3. $\mathcal{P}$ fetches the next unused $m-1$ authenticated values and denotes them as $([\![r_1]\!], \ldots, [\![r_{m-1}]\!])$. For $j \in [m-1]$, $\mathcal{P}$ computes $z_j = x_j \cdot y_j$ and sends $b_j = z_j - r_j$ to verifiers. Verifiers compute $[\![z_j]\!] = b_j + [\![r_j]\!]$.
4. Parties compute $[\![z_m]\!] = [\![z]\!] - \sum_{j=1}^{m-1} [\![z_j]\!]$.
5. $\mathcal{P}$ fetches the next unused m authenticated values and denotes them as $([\![\tilde{r}_1]\!], \ldots, [\![\tilde{r}_m]\!])$. For $j \in [m]$, $\mathcal{P}$ computes $d_j = f_i(m+j+1) \cdot g_i(m+j+1)$ and sends $c_j = d_j - \tilde{r}_j$ to verifiers. Verifiers compute $[\![d_j]\!] = c_j + [\![\tilde{r}_j]\!]$.
6. All parties interpolate a degree-$2m$ polynomial $[\![h(\cdot)]\!]$ such that $h(j) = z_j$ for $j \in [m+1]$ and $h(j) = d_{j-m}$ for $j \in [m+2, 2m+1]$.
7. Parties compute
   $$\eta \leftarrow \mathsf{H}(\mathsf{IP}, m, \eta_{\mathsf{prev}}, \{\hat{b}_j\}_{j=1}^{3}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m}),$$
   in which the first two parameters are used as the identifier and h_{prev} is the hash result from the previous invocation of H. If $\eta \in [m]$, all parties abort. Parties output $([\![a]\!], [\![b]\!], [\![c]\!])$ such that $a = f(\eta)$, $b = g(\eta)$ and $c = h(\eta)$.
8. $\mathcal{P}$ sends a, b, c to all verifiers.

All verifiers use the authentication to check that $\mathcal{P}$ reveals the correct reconstruction results and check whether (a, b, c) satisfy the multiplication relation. The security is due to the random multi-

plication triple shared by the prover in the last round of the dimension reduction. We present the whole protocol for verifying an inner-product tuple in π_{VerifyIP}. The number of random authenticated additive sharings in $\mathcal{K}$ we need is $c = 2\nu(m-1)+4$, where $\nu = \lceil \log_m N \rceil$.

Procedure 3: π_{VerifyIP}

Input: Authenticated additive secret sharings of an inner product triple $(\llbracket \boldsymbol{x} \rrbracket, \llbracket \boldsymbol{y} \rrbracket, \llbracket z \rrbracket)$ of dimension N such that $z = \sum_{i=1}^{N} x_i \cdot y_i$, the compression parameter m.
Verify:

1. $\mathcal{P}$ and $\mathcal{V}_1, \ldots, \mathcal{V}_n$ invoke the Procedure $\pi_{\mathsf{VerifyIPProc}}$ with input $(\llbracket \boldsymbol{x} \rrbracket, \llbracket \boldsymbol{y} \rrbracket, \llbracket z \rrbracket)$ and compression parameter m, and recursively invoke it with the output from the previous invocation, until the output dimension is less than or equal to m.
2. Verifiers invoke the Procedure $\pi_{\mathsf{VerifyIPFinal}}$ which outputs a single multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ together with (a, b, c).
3. For $i \in [n]$, P_i constructs $\langle \theta_1^i \rangle, \langle \theta_2^i \rangle, \langle \theta_3^i \rangle$ such that $\theta_1^i = \theta_2^i = \theta_3^i = 0$. It distributes shares to other verifiers. This step can be done in the preprocessing phase.
4. All verifiers compute $\langle o_1 \rangle = \langle \Delta \cdot a \rangle - a \cdot \langle \Delta \rangle + \sum_{i=1}^{n} \langle \theta_1^i \rangle$, $\langle o_2 \rangle = \langle \Delta \cdot b \rangle - b \cdot \langle \Delta \rangle + \sum_{i=1}^{n} \langle \theta_2^i \rangle$, and $\langle o_3 \rangle = \langle \Delta \cdot c \rangle - c \cdot \langle \Delta \rangle + \sum_{i=1}^{n} \langle \theta_3^i \rangle$. Then for $i \in [n]$, P_i invokes $\mathcal{F}_{\mathsf{Commit}}$ with input o_1^i, o_2^i, o_3^i.
5. Verifiers check that they receive the same messages from $\mathcal{P}$ and the same commitments. This is done by letting each verifier compute a hash of his received messages and send the hash result to all other verifiers. If a verifier receives different hash values from other verifiers, this verifier aborts.
6. All verifiers open the above commitments. If $\sum_{i=1}^{n} o_1^i = \sum_{i=1}^{n} o_2^i = \sum_{i=1}^{n} o_3^i = 0$ and $c = a \cdot b$, all verifiers accept. Otherwise, all verifiers abort.

4.2 Online Protocol

Protocol 3: Π_{Online}

Define an arithmetic circuit $\mathcal{C}$ with $|\mathcal{I}|$ input wires and $|\mathcal{C}|$ multiplication gates. Let $N = |\mathcal{I}| + |\mathcal{C}|$, m be the compression parameter, and $c = 2\lceil \log_m N \rceil (m-1) + 4$ be the number of random authenticated additive sharings in $\mathcal{K}$ needed during the inner-product check. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\kappa$ be a random oracle. Let $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^*$ be a pseudorandom generator.

1. **Preprocess:** $\mathcal{P}$ and $\mathcal{V}_1, \ldots, \mathcal{V}_n$ call Π_{Prep} to prepare N random authenticated additive sharings in $\mathcal{R}$ and c random authenticated additive sharings in $\mathcal{K}$. For all $j \in [d]$ and $\ell \in [\sigma]$, all verifiers hold $[\Delta_j | \ell]_t$. In addition, each verifier $\mathcal{V}_i$ holds nonce_i and $\mathcal{P}$ holds $\mathsf{nonce}_1, \ldots, \mathsf{nonce}_n$.
2. **Distribute Circuit Wire Values:** For each group of $\sigma \cdot d$ wires, let $\{v_{i,j}\}_{i \in [\sigma], j \in [d]}$ denote the wire values to be shared by $\mathcal{P}$.
   (a) $\mathcal{P}$ fetches the first $\sigma \cdot d$ unused random authenticated additive sharings in $\mathcal{R}$, denoted by $\{\llbracket r_{i,j} \rrbracket\}_{i \in [\sigma], j \in [d]}$. $\mathcal{P}$ computes $\lambda_{i,j} = v_{i,j} - r_{i,j}$ for all $i \in [\sigma], j \in [d]$. Then $\mathcal{P}$ views $\boldsymbol{\lambda}_i = (\lambda_{i,1}, \ldots, \lambda_{i,d})$ as an element in $\mathcal{K}$ and computes a degree-$(\sigma-1)$ packed Shamir sharing of $(\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_\sigma)$, denoted by $[\boldsymbol{D}]_{\sigma-1}$. $\mathcal{P}$ distributes the shares of $[\boldsymbol{D}]_{\sigma-1}$ to all verifiers.
   (b) All verifiers locally convert $[\boldsymbol{D}]_{\sigma-1}$ to additive sharings $\langle \boldsymbol{\lambda}_1 \rangle, \ldots, \langle \boldsymbol{\lambda}_\sigma \rangle$, and then convert these k additive sharings in $\mathcal{K}$ to $\sigma \cdot d$ additive sharings $\{\langle \lambda_{i,j} \rangle\}_{i \in [\sigma], j \in [d]}$ in $\mathcal{R}$.
   (c) For all $i \in [\sigma], \ell \in [d]$, all verifiers locally compute $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell | i]_t$ and locally convert it to additive sharings $\{\langle \lambda_{i,j} \cdot \Delta_\ell \rangle\}_{j \in [d]}$ in $\mathcal{R}$. Then for all $i \in [\sigma], j \in [d]$, all verifiers locally convert $\{\langle \lambda_{i,j} \cdot \Delta_\ell \rangle\}_{\ell \in [d]}$ to an additive sharing $\langle \lambda_{i,j} \cdot \Delta \rangle$ in $\mathcal{K}$.
   (d) All verifiers locally compute $\llbracket v_{i,j} \rrbracket = \llbracket r_{i,j} \rrbracket + \llbracket \lambda_{i,j} \rrbracket$ for all $i \in [k], j \in [d]$.
3. **Procedure for Fiat-Shamir:**
   (a) For $i \in [n]$, $\mathcal{P}$ computes $\mathsf{com}_i = \mathsf{H}(\mathcal{V}_i, \boldsymbol{s}^i, \mathsf{nonce}_i)$ where $\boldsymbol{s}^i$ are the shares distributed to $\mathcal{V}_i$ in Step 2.
   (b) $\mathcal{P}$ sends $(\mathsf{com}_1, \ldots, \mathsf{com}_n)$ to all verifiers.
   (c) For $i \in [n]$, $\mathcal{V}_i$ verifies com_i. If the check fails, it aborts.
   (d) All verifiers check that they receive the same $(\mathsf{com}_1, \ldots, \mathsf{com}_n)$ from $\mathcal{P}$. This is done by letting each verifier compute a hash of his received messages and send the hash result to all other verifiers. If a verifier receives different hash values from other verifiers, this verifier aborts.
   (e) All parties compute $s = \mathsf{H}(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$, where the first parameter is used as the identifier.
4. **Verification of Multiplications:** All verifiers follow the circuit structure and compute an authenticated additive sharings for all wire values. Note that the missing wires are all from the outputs

of addition gates, which can be computed locally from the input authenticated additive sharings. Let m_1 denote the number of multiplication gates. For $\ell \in [m_1]$, denote $(\llbracket v_{\alpha_\ell} \rrbracket, \llbracket v_{\beta_\ell} \rrbracket, \llbracket v_{\gamma_\ell} \rrbracket)$ as the shares of input and output wire values of the ℓ-th multiplication gate.

(a) Parties compute uniform coefficients $(\chi_1, \ldots, \chi_{m_1}) \leftarrow \mathsf{PRG}(s)$. They construct an inner product triple

$$\llbracket \boldsymbol{x} \rrbracket = (\chi_1 \cdot \llbracket v_{\alpha_1} \rrbracket, \ldots, \chi_{m_1} \cdot \llbracket v_{\alpha_{m_1}} \rrbracket)$$
$$\llbracket \boldsymbol{y} \rrbracket = (\llbracket v_{\beta_1} \rrbracket, \ldots, \llbracket v_{\beta_{m_1}} \rrbracket)$$
$$\llbracket z \rrbracket = \sum_{i=1}^{m_1} \chi_i \cdot \llbracket v_{\gamma_i} \rrbracket$$

(b) Recall that m is the compression parameter. If the dimension of the vectors $(\llbracket \boldsymbol{x} \rrbracket, \llbracket \boldsymbol{y} \rrbracket)$ is not multiple of m, the verifiers pad zeros at the end of $(\llbracket \boldsymbol{x} \rrbracket, \llbracket \boldsymbol{y} \rrbracket)$ until their dimension is $m \cdot \lceil \frac{m_1}{m} \rceil$.

(c) Verifiers invoke π_{VerifyIP} to check $(\llbracket \boldsymbol{x} \rrbracket, \llbracket \boldsymbol{y} \rrbracket, \llbracket z \rrbracket)$. If all verifiers accept the check, they output accept.

Theorem 2. *Assume that a random function from the class of expansion functions in $\mathcal{F}_{\mathsf{nVOLE}}$ is a PRG. Protocol Π_{Online} securely realizes $\mathcal{F}_{\mathsf{ZK}}$ in the presence of a malicious adversary $\mathcal{A}$ who can corrupt $\mathcal{P}$ and up to t-out-of-n verifiers among $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ in the $(\mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Commit}})$-hybrid and random oracle model.*

We refer the readers to Section C.2 for the proof.

Remark 5. Unlike the previous honest-majority MVZK protocol by Yang and Wang [YW22], we do not assume the existence of a broadcast channel to let the prover distribute the extended witness and verification material to the parties. Note that, for security with abort, broadcast channels can be easily instantiated over point-to-point channels by having the parties cross-check that they receive the same message by sending the hash of the message to each other. This is the approach we *implicitly* take in our work. If we were to assume a broadcast channel, we may remove the extra rounds required for cross-checking (which would be "hidden" in the instantiation of the broadcast channel), but this would not save communication since the number of downloaded bits by each party would be the same.

5 Performance Study

In this section we study the performance of our protocol, and compare it with respect to related work. Recall that our multiverifier ZK construction supports any number of active corruptions $t < n(1-\epsilon)$, for *any* constant $0 < \epsilon < 1$. Since in the honest majority regime where $\epsilon > 1/2$ one can design protocols based only on information-theoretic primitives such as threshold secret-sharing (*e.g.* [YW22; Bau+22a]), without resotring to public-key operations such as VOLE as in here, we are particularly interested in the performance of our construction in the dishonest majority context, in which $\epsilon \leq 1/2$. In this case, the only related work in the context of multiverifier ZK is [Zho+23], which as we will see is mostly of theoretical interest, with our work outperforming theirs but at least two orders of magnitude. First, in Section 5.1 we study the communication complexity of our protocol and compare it to related works. Finally, in Section 5.2 we present experimental results derived from our end-to-end implementation.

5.1 Communication Complexity

We first study the communication costs of our protocol. Since all related works in multiverifier ZK are set over fields, we focus in this case for this analysis, and we assume for simplicity that $|\mathbb{F}| > 2^\kappa$. For measuring communication we ignore costs that are sublinear on the circuit size $|\mathcal{C}|$ **or** the number of verifiers n. This typically includes calls to random beacon functionalities, amortized consistency checks, recursive checks, Fiat-Shamir hashes, etc. We let $|\mathcal{I}|$ be the number of inputs and $|\mathcal{C}|$ the size of the circuit, measured in number of multiplications.

First, we measure the communication of our protocol ($\Pi_{\mathsf{Online}} + \Pi_{\mathsf{Prep}}$). Our preprocessing Π_{Prep} requires $M = \lceil (|\mathcal{I}| + |\mathcal{C}| + c \cdot d)/\sigma \rceil$ VOLE pairwise correlations from $\mathcal{F}_{\mathsf{nVOLE}}$, where $\sigma = n - t = \epsilon n$ is the number of honest parties, and c is the number of additive sharings needed for the consistency check, which is sublinear in $|\mathcal{C}|$ and hence we ignore it. Generating each of these using Π_{nVOLE} corresponds (ignoring the cost of the consistency check, which is independent of $|\mathcal{C}|$) to each pair of parties calling $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with length $\approx M$. Using our instantiation Wolverine-based protocol $\Pi_{\mathsf{VOLE}}^{\mathsf{prog}}$ that appears in the full version of this work, this cost is sublinear in M, and hence we ignore it for the purpose of the estimation of the communication costs.[8] For the online phase, $\mathcal{P}$ sends to *each* verifier $(|\mathcal{I}| + |\mathcal{C}|)/\sigma$ elements, for a *total* of $n(|\mathcal{I}| + |\mathcal{C}|)/(\epsilon n) = (|\mathcal{I}| + |\mathcal{C}|)/\epsilon$ elements.

Crucially, we see that for a fixed ϵ, communication is constant *independently of* n, which is an essential property for scalability, and is one of the major selling points of our construction. Furthermore, the more honest parties there are, that is, the larger that ϵ becomes, the better this constant is. We see then that our protocol benefits from having more honest parties, even without reaching honest majority. Let us compare now to [Zho+23]. This protocol is designed for maximal adversary dishonest majority. For a fair comparison, we take the number of verifiers in their work to be $n' = t + 1 \approx n(1 - \epsilon)$, since they only need one honest verifier to ensure security. The preprocesssing in Π_{SIF} in [Zho+23] is based on that from the BDOZ protocol [Ben+11], which the authors assume as a black-box. This uses semi-homomorphic encryption and zero-knowledge proofs, which are more expensive than VOLE, and yet for this comparison we omit their communication costs. Still, the offline phase in [Zho+23] requires $\mathcal{P}$ to learn the "masks" after BDOZ has been executed (protocol Π_{Prep} in [Zho+23, Fig. 5]), which requires *each* verifier to send $3|\mathcal{C}| + |\mathcal{I}|$ elements to $\mathcal{P}$. For the online phase, $\mathcal{P}$ sends $|\mathcal{I}| + 2|\mathcal{C}|$ elements to *each* verifier, so $n'(|\mathcal{I}| + 2|\mathcal{C}|)$ *total*. Then, for the verification the verifiers need to reconstruct to each other $2|\mathcal{C}|$ elements.[9] This is done in [Zho+23] by everyone sending to everyone in one round, since their work is mostly interested in minimizing round count. This is quadratic in n'. Removing the round requirement, this can be improved to $4n'|\mathcal{C}|$ by using an intermediate king for reconstruction, increasing their rounds by 1. We use this for our communication estimates (which plays in their favor).

Summing up, the communication for Π_{SIF} [Zho+23] is $n(1 - \epsilon)(7|\mathcal{C}| + |\mathcal{I}|)$. Note that, for constant ϵ, communication is *linear* in n, with a high constant multiplying the circuit size. Furthermore, larger ϵ only helps [Zho+23] reduce the "slope" of the linear growth. In Section 5.2 we present experimental results for our protocol, but note that we do not compare experimentally with [Zho+23], which is mostly of theoretical interest. Their linear growth, coupled with the fact that [Zho+23] relies on computationally more expensive techniques than ours (HE and generic NIZKs used in [Ben+11] in contrast to recent optimized techniques for VOLE as in ours), are enough evidence that our protocol drastically outperforms that of [Zho+23]. In Table 1 we present some concrete communication costs for our protocol with respect to [Zho+23] (and also other works), which puts this massive gap in evidence. We discuss this more thoroughly below.

Finally, we remark that in terms of *rounds* of communication, our protocol stands well. In the offline phase, the verifiers only need to receive the Expand function and the seeds from $\mathcal{P}$, and they interact with each other to setup the VOLE correlations required by Π_{Prep}. With our Wolverine-based instantiation of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, discussed in the full version of this work, which is the one we use in our end-to-end implementation, we obtain 6 rounds in total, including sending the nonces to $\mathcal{P}$. The online phase only requires $\mathcal{P}$ to send the proof, followed by two round of interactions[10] between the verifiers to commit-and-open in order to check its correctness. This round complexity is comparable to that from [Zho+23]: they assume BDOZ [Ben+11] preprocessing for the offline phase, which already adds a few rounds, and they add one more on top for their offline phase. In the online phase then they have only two rounds, although, as we discussed before, this would lead to a communication cost of $\Omega(n^2|C|)$. For linear communication (which is already not very good concretely, as reported in Table 1), one extra round must be added, matching ours.

[8] This is only for communication, since later in Section 5.2 we discuss our end-to-end implementation which includes the full instantiaton of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ as well as the sublinear check.

[9] We note that our protocol only involves sublinear communication among verifiers (and in fact the same holds for [Bau+22a; YW22], which are honest majority).

[10] This is achieved if assuming a broadcast channel, and 3 rounds otherwise. We note that [Zho+23] also assumes a broadcast channel.

Comparison in the honest majority regime As we have mentioned, the honest majority setting enables for more lightweight techniques that do not use public key cryptography, and this is exploited by the work of Baum et al. [Bau+22a], which proposes two protocols for $t < n/4$ and $t < n/3$, and Yang and Wang [YW22], which presents a protocol for $t < n(1-\epsilon)$ but only for $\epsilon \in (1/2, 1)$ (that is, honest majority). However, as have mentioned, our protocol works for $t < n(1-\epsilon)$ for any ϵ, including honest majority $\epsilon \in (1/2, 1)$. Hence, even though this is not our original target regime, it is still meaningful to study the communication of our protocol with respect to these in [YW22; Bau+22a]. Towards this end, let us first study the communication of these protocols.

- Π_{4t} **in [Bau+22a],** $t < n/4$**.** This protocol does not exploit the gap ϵ, so we use the minimum amount of verifiers required for the desired ratio: $n' \approx 4t = 4n(1-\epsilon)$. In the offline phase, *each* verifier sends $\approx n' \cdot \frac{|\mathcal{C}|+|\mathcal{I}|}{3/4 \cdot n'} = \frac{4}{3}(|\mathcal{C}| + |\mathcal{I}|)$ shares to all other verifiers.[11] In the online phase, $\mathcal{P}$ sends the proof to *each* verifier whose size is $|\mathcal{C}| + |\mathcal{I}|$ elements. The cost of the verification is constant in $|\mathcal{C}|$, hence we ignore it. Overall, multiplying by n', the total is $\frac{28}{3}n(1-\epsilon)(|\mathcal{C}| + |\mathcal{I}|)$
- Π_{3t} **in [Bau+22a],** $t < n/3$**.** Here we take $n' \approx 3t = 3n(1-\epsilon)$. In the offline phase, *each* verifier sends $\approx n' \cdot \frac{|\mathcal{C}|+|\mathcal{I}|}{2/3 \cdot n'} = \frac{3}{2}(|\mathcal{C}| + |\mathcal{I}|)$ elements. In the online phase, $\mathcal{P}$ sends the proof to *each* verifier whose size is $|\mathcal{C}| + |\mathcal{I}|$. The cost of the verification is logarithmic in $|\mathcal{C}|$, hence we ignore it. Overall, multiplying by n', the total is $\frac{15}{2}n(1-\epsilon)(|\mathcal{C}| + |\mathcal{I}|)$
- $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$ **in [YW22],** $t \approx n(1-\epsilon)$ **for** $\epsilon \in (1/2, 1)$**.** For the proof, $\mathcal{P}$ sends $\approx (|\mathcal{C}| + |\mathcal{I}|)/((\epsilon - 1/2)n)$ shares to *each* verifier.[12] The circuit verification cost is $O(n \log |\mathcal{C}|)$ and hence we omit it. The total is then $(|\mathcal{C}| + |\mathcal{I}|)/(\epsilon - 1/2)$.

In Table 1, we present the communication of these protocols and ours, including the one from [Zho+23], for a threshold $t \approx n(1-\epsilon)$, for different values of ϵ and n, and a circuit of size 2^{30} with 2^{20} inputs over a 64-bit field. Note that Π_{4t} only supports $\epsilon > 3/4$, Π_{3t} only supports $\epsilon > 2/3$, and $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$ only supports $\epsilon > 1/2$; we write "N/A" for values outside these ranges. We note several key points from this table. As we noted already, we see the results in [Zho+23] are mostly of theoretical interest: for $n = 25$ and 10% honest parties, [Zho+23] requires already 1.4 TB of communication while we only need 86 GB, and for $n = 200$ this is already 10.8 TB vs the same 86 GB for us. Now, note that the communication of [YW22], as ours, does not increase with n for a fixed ϵ. However, the underlying constants are better in our case: our communication is roughly half the one from [YW22], which stems from the fact that we can "pack" twice as many secrets in packed secret-sharing. This is because we leverage VOLE for products instead of Shamir multiplicativity as in [YW22], which requires a smaller polynomial degree and hence a worse packing parameter. Instead, in our work we are able to use a maximal degree of $n - 1$. Finally, communication with respect to the protocols in [Bau+22a] is even better: their communication scales with n, while ours remains constant, and for example for 80% honest parties out of $n = 25$, the communication in Π_{3t} from [Bau+22a] takes ≈ 400 GB, while ours only takes 10.7 GB. For $n = 200$ ours takes (approximately) the same 10.7 GB, while Π_{3t} increases to 3.2 TB.

These results show that our protocol not only presents the most efficient multiverifier ZK protocol in the dishonest majority setting, but it also has the potential to drastically improve the performance over prior works in the honest majority regime, which is a very interesting byproduct of our work given that our main target is the dishonest majority case. We already see from Table 1 massive improvements in terms of communication. We leave it to future work to explore whether our techniques can be also beneficial in the honest majority regime in terms of end-to-end runtimes, given the nature of the tools used in our work (*e.g.* VOLE) being different than these by [YW22; Bau+22a], which are specifically optimized for honest majority (*e.g.* Shamir secret-sharing). We analyze the comparison between [Bau+22a] and our work in Section 5.2. As an additional note, we observe that the work of [YW22] has not been implemented and so we find it harder to provide an estimate.

[11] Each share is signed, which adds communication. We ignore this, which plays in their favor.

[12] This hides extra (constant, but not necessarily small) factors in [YW22] arising from circuit transformation, needed to accommodate for the use of packed secret-sharing. We avoid this in our work since we only used packing for compressing the proof, but for verification we rely on additive secret-sharing.

n	ϵ	Π_{3t} [Bau+22a]	Π_{4t} [Bau+22a]	$\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$ [YW22]	Π_{SIF} [Zho+23]	**Ours**
25	0.9	200.6 GB	161.2 GB	21.5 GB	150.3 GB	9.6 GB
	0.8	401.3 GB	322.4 GB	28.7 GB	300.7 GB	10.7 GB
	0.6	N/A	N/A	86.0 GB	601.4 GB	14.3 GB
	0.4	N/A	N/A	N/A	902.1 GB	21.5 GB
	0.2	N/A	N/A	N/A	1.2 TB	43.0 GB
	0.1	N/A	N/A	N/A	1.4 TB	86.0 GB
50	0.9	401.3 GB	322.4 GB	21.5 GB	300.7 GB	9.6 GB
	0.8	802.5 GB	644.9 GB	28.7 GB	601.4 GB	10.7 GB
	0.6	N/A	N/A	86.0 GB	1.2 TB	14.3 GB
	0.4	N/A	N/A	N/A	1.8 TB	21.5 GB
	0.2	N/A	N/A	N/A	2.4 TB	43.0 GB
	0.1	N/A	N/A	N/A	2.7 TB	86.0 GB
100	0.9	802.5 GB	644.9 GB	21.5 GB	601.4 GB	9.6 GB
	0.8	1.6 TB	1.3 TB	28.7 GB	1.2 TB	10.7 GB
	0.6	N/A	N/A	86.0 GB	2.4 TB	14.3 GB
	0.4	N/A	N/A	N/A	3.6 TB	21.5 GB
	0.2	N/A	N/A	N/A	4.8 TB	43.0 GB
	0.1	N/A	N/A	N/A	5.4 TB	86.0 GB
200	0.9	1.6 TB	1.3 TB	21.5 GB	1.2 TB	9.6 GB
	0.8	3.2 TB	2.6 TB	28.7 GB	2.4 TB	10.7 GB
	0.6	N/A	N/A	86.0 GB	4.8 TB	14.3 GB
	0.4	N/A	N/A	N/A	7.2 TB	21.5 GB
	0.2	N/A	N/A	N/A	9.6 TB	43.0 GB
	0.1	N/A	N/A	N/A	10.8 TB	86.0 GB

Table 1. Communication costs of our protocol with respect to related works for a circuit of size $|\mathcal{C}| = 10^9$ with $|\mathcal{I}| = 10^6$ inputs. n is the total number of verifiers, and ϵ is the fraction of honest verifiers. "N/A" means the given protocol does not support such high corruption thresholds.

5.2 Implementation and Experiments

We evaluate the performance of an end-to-end C++ implementation of our multi-verifier zero-knowledge proof protocol over finite fields. The implementation is based on EMP-toolkits [WMK16] and Intel HEXL [Boe+21]. The polynomial interpolation and evaluation are optimized by the number theoretic transform (NTT) provided by Intel HEXL, and we use an NTT-friendly field $p = 2^{59} - 2^{28} + 1$. All experiments are executed in AWS EC2 c5.metal machines with 96 vCPU and 192 GiB RAM unless otherwise specified. We emulate the throttled network bandwidth and latency using the Linux netem tool. For experiments with ≤ 32 verifiers, all parties reside at 1 machine and are executed by different processes. For larger-scale experiments, we split verifiers into multiple machines due to memory constraints. The verifier implementation utilizes two threads for pair-wise VOLE protocol execution, which prevents parties from staying idle.

End-to-end performance and parameters We benchmark the performance of our protocol with various combinations of (n, σ), and σ/n is the ratio of honest verifiers. To utilize the optimization by NTT, we choose both (n, σ) to be powers of 2. For each parameter set, we fix the bandwidth to be 1Gbps and evaluate an arithmetic circuit of 2^{30} multiplication gates. Table 2 shows the throughput (million gates per second) of the protocol with these parameter choices. When fixing the number of parties, the throughput increases if σ increases. This is because both computation and communication overhead are inversely proportional to σ. On the other hand, if the ratio of honest parties σ/n is fixed, the throughput is almost unchanged no matter how large (n, σ) are. This property enables the protocol to scale to a large number of parties because its efficiency does not degrade with the increase of n.

σ/n	4	8	16	32	64
50%	1.51	1.59	1.61	1.43	1.47
25%	-	0.94	0.97	0.83	0.88
12.5%	-	-	0.55	0.45	0.49
6.25%	-	-	-	0.25	0.27

Table 2. Throughput ($\times 10^6$ gates/second) of MVZK protocol in 1Gbps network. The 1st row indicates the number of parties.

Performance with network emulation We test the throughput of our protocol in different network conditions and show the results in Table 3. In the first experiment, the bandwidth is throttled to 5Gbps, 1Gbps, 500Mbps and 100Mbps, respectively. We execute our ZKP with 32 verifiers and evaluate a circuit of 2^{30} gates. The communication overhead is dominated by $O(n|C|/\sigma)$ field elements sent from $\mathcal{P}$ to verifiers. When bandwidth is larger than 500Mbps, the throughput only slightly degrades with the decrease of bandwidth. In the second experiment, the bandwidth is fixed to 1Gbps, and we simulate the network latency of 2, 10, 30, and 60ms. Due to the constant-round feature of our protocol, the performance is not impacted by the network delay.

Bandwidth	5Gbps	1Gbps	500Mbps	100Mbps
Throughput	1.18	0.84	0.72	0.18
Latency	2ms	10ms	30ms	60ms
Throughput	0.83	0.82	0.80	0.80

Table 3. Throughput ($\times 10^6$ gates/second) of MVZK protocol in different network settings.

Scalability and memory overhead The memory overhead of our protocol is linear to the number of verifiers, and is proportional to the memory usage when verifying the statement in plaintext. Hence the protocol is able to achieve high scalability by evaluating the circuit chunk-by-chunk instead of processing the whole circuit in memory. We demonstrate this feature by a proof of matrix multiplication: for public matrix $C \in \mathbb{F}_p^{d\times d}$, $\mathcal{P}$ proves the knowledge of two private matrices $A, B \in \mathbb{F}_p^{d\times d}$ such that $A \cdot B = C$. We use a native matrix multiplication circuit of size d^3 gates.

By setting $(n, \sigma) = (32, 8)$, bandwidth 1Gbps and network latency 60ms, we show the running time and memory usage of the proof with varying matrix dimensions in Table 4. The running time is approximately linear to the circuit size (d^3) and the memory overhead is minorly affected by the increase of d. Indeed, most memory consumption stems from storing VOLE correlations. The verifiers need to store additional authentication messages; thus, their RAM usage is higher than $\mathcal{P}$. Furthermore, the memory overhead can be reduced by adjusting the LPN parameters, which saves the memory but increases the communication overhead. In real-world scenarios, it offers a flexible choice in terms of the trade-off between memory and communication overhead.

Comparison to previous MVZK The prior MVZK proposed by Baum et al. [Bau+22a] lies in the honest-majority regime and is implemented for small fields. Results in [Bau+22a, Table 2] show that for 7 verifiers and $1/3$ corruptions, it takes 326.48ms for end-to-end protocol execution when proving million-size circuits, which results in a comparable throughput of 3×10^6gates/s. Our closest result is the case of 8 verifiers and 50% corruption, which achieves 1.59Mg/s. We estimate that if our protocol is reduced to the same number of corruption and field size, its communication overhead will be further reduced by $32\times$. Meanwhile, [Bau+22a, Table 3] shows that for $n = 100$ and $1/3$ corruptions, its protocol Π_{3t} results in a throughput of 0.67×10^6gates/s. Though our experiments do not scale to $n = 100$, we conclude from Table 2 that once the ratio σ/n is fixed, the running time of our protocol does not degrade with the increase of n. Hence we estimate the throughput of our protocol still maintains $> 1.4 \times 10^6$gates/s when scaling from 64 to 100 parties, which performs better than it in [Bau+22a] even with more corruptions.

We state the dramatic advantages of our protocol against Zhou et al. [Zho+23] in Section 5.1 by showing the estimated gap in communication in Table 1. The benchmark by Zhou et al. only scales to 7 verifiers. Its throughput is around 0.03×10^6gates/s (estimated from [Zho+23, Table 3]), while our protocol achieves throughput around 0.5×10^6gates/s with even higher corruption threshold.

Comparison to interactive ZKP We compare the running time of our MVZK with Quicksilver [Yan+21], which is a single-verifier VOLE-ZK with low memory overhead and high efficiency. In the scenario when there are σ honest verifiers, we assume $\mathcal{P}$ proves the same statement to $n - \sigma + 1 = 25$ verifiers sequentially, which guarantees that it interacts with at least 1 honest verifier.

We ignore the polynomial optimization of Quicksilver and only test for generic circuit computation. Its running time for the same tasks is shown in the last column of Table 4. Comparing two protocols, our MVZK decreases the running time by $3.3\times$-$4.4\times$ and communication overhead by $6.25\times$.

Comparison to IOP-based implementations Recent development of IOP-based proof systems (e.g., Brakedown [Gol+23], Orion [XZS22] and Ligetron [WHV24]) show high efficiency and low memory overhead compared to other SNARKs. While none of these implementations are open-sourced, we compare our MVZK with these schemes by the prior benchmarks provided by [WHV24, Figure 6e-6f]. It shows that the throughput (prover efficiency) of these ZKP are in the range of $0.3 - 1.6 \times 10^6$gates/s. In a machine with 16GB memory size, these ZKPs are able to process circuits of $2^{22} - 2^{25}$ gates. [WHV24, Figure 6f] also shows that the memory overhead of these schemes are nearly linear to the size of the circuit. While some of these banchmarks enable multi-threading, our MVZK achieves throughput close to these scheme with a single thread (Table 2). Meanwhile, our MVZK preserves the memory efficiency of interactive ZKP with a flat memory overhead even for billion-size circuits (Table 4). Note that we can not compare with the garbage collection-enabled Ligetron because its performance for large circuits ($>2^{25}$ gates) is unknown.

Comparison to distributed zk-SNARK EOS [Chi+23] is a private zkSNARK delegation scheme that outsources the proof generation to a set of workers without revealing the witness. It improves the proof generation time and scalability of zkSNARK. It provides a stronger functionality than designated MVZK since its workers produce public-verifiable proofs that can be verified by anyone. We compare the performance of EOS and MVZK for the task of solely proving a statement to a fixed set of verifiers. According to the performance report of EOS, it takes around 1000s for two workers to evaluate a statement of size 2^{25} under a 3Gbps network. This number excludes the preprocessing time. Using the same time budget, our end-to-end MVZK is able to prove to 32 verifiers with 75% corruption of a circuit of size 2^{30}, or with 93.75% corruption of a circuit of size 2^{28}. Note that each EOS worker runs in a machine with 192GB RAM and 96 cores, while all of our verifiers only utilize 192GB RAM and 96 cores in total.

Dim.	Time ($\times 10^3$s)	Memory (GB)		[Yan+21] ($\times 10^3$s)
		Prover	Per Verifier	
512	0.22	2.69	5.67	0.73
768	0.58	2.70	5.70	2.46
1024	1.34	2.72	5.73	5.82
1280	2.56	2.73	5.77	11.38

Table 4. Running time and memory overhead of our MVZK protocol for the proof of matrix multiplication.

In the following, we compare our MVZK with other relevant works of which a proper publicly-available benchmark is unattainable. Their implementations are either hardcoded to prove very specific statements (e.g., Prio SNIP for secure aggregation [CGB17] and VOLE-in-the-Head for PQ signatures [Bau+23]), or their practical instantiation is not in the same security model as ours (*e.g.*, distributed ZK [Boy+19b]).

Comparison with Prio SNIP The secret-shared non-interactive proofs (SNIPs) proposed in Prio [CGB17] can be viewed as a multi-verifier ZK. If we consider that Prio SNIP is equipped with optimizations proposed by its follow-up works [Bon+19; Boy+20], our MVZK shares the same asymptotic communication and computation complexity with Prio. However, our MVZK preserves the soundness property against malicious adversaries who corrupt both the prover and up to $t < n$ verifiers, while Prio does not tolerate prover-verifier collusion. Meanwhile, although we can use our techniques to endow Prio with security against prover-verifier collusion, it would result in $O(n|C|)$ communication even if $t < (1 - \epsilon)n$ for a constant ϵ, while our MVZK only needs $O(|C|)$ in this case.

Comparison with distributed ZK Even though the term Distributed ZK (DZK) itself is not very well defined, it generally refers to the fully linear IOP (FLIOP) approach proposed in [Bon+19; Boy+20; Boy+19b; Boy+21], used in the context of proving statements over secret-shared data. One of the core technical contributions of our work is enabling a prover to secret-share efficiently an extended witness over a set of verifiers. Once this is done, the check the verifiers engage in is not very different from that in DZK, and in fact, part of our verification protocol $\pi_{\mathsf{VerifyIPProc}}$ shown in Figure 3 is inspired by FLIOPs. However, our construction specifically leverages the fact that the prover $\mathcal{P}$, who holds the witness, has established certain VOLE correlations with the verifiers, which enables secret-sharing not only the extended witnesses but also the messages in the FLIOP much more efficiently than in other instantiations of the FLIOP model. Our work can be used in the scenario of secure aggregation, client-server model MPC, and blockchain oracles to let a client prove the input validity.

Comparison with VOLE-in-the-Head VOLE-in-the-Head (VitH) is a NIZK built upon Quicksilver with $O(|C|)$ proof size [Bau+23]. When using VitH to realize MVZK, each verifier still needs to receive an $O(|C|)$-size proof, which results in $O(n|C|)$ communication overhead in total, whereas ours is only $O(|C|)$. Moreover, our protocol enjoys streaming properties and thus has better memory efficiency and scalability because of the streaming setting. Experiments in Table 4 show that our MVZK has a flat memory usage with the increase of circuit size. On the other hand, VitH is a one-shot protocol with memory overhead linear to the circuit size. Thus, VitH would need a large machine for billion-size circuits benchmarked in our experiments. Our scheme can easily adapt to a smaller RAM budget by adjusting the VOLE parameters.

Acknowledgments

Y. Song was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

This paper was prepared in part for information purposes by the Artificial Intelligence Research Group and the AlgoCRYPT CoE of JPMorgan Chase & Co and its affiliates ("JP Morgan") and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy, or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer, or solicitation for the purchase or sale of any security, financial instrument, financial product, or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

[Abs+19] Mark Abspoel et al. "Efficient Information-Theoretic Secure Multiparty Computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois Rings". In: 2019, pp. 471–501. DOI: 10.1007/978-3-030-36030-6_19.

[Abs+20] Mark Abspoel et al. "Asymptotically Good Multiplicative LSSS over Galois Rings and Applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$". In: 2020, pp. 151–180. DOI: 10.1007/978-3-030-64840-4_6.

[ACF02] Masayuki Abe, Ronald Cramer, and Serge Fehr. "Non-interactive Distributed-Verifier Proofs and Proving Relations among Commitments". In: 2002, pp. 206–223. DOI: 10.1007/3-540-36178-2_13.

[Add+22] Surya Addanki et al. "Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares". In: *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*. Ed. by Clemente Galdi and Stanislaw Jarecki. Vol. 13409. Lecture Notes in Computer Science. Springer, 2022, pp. 516–539. DOI: 10.1007/978-3-031-14791-3_23. URL: https://doi.org/10.1007/978-3-031-14791-3_23.

[AKP22] Benny Applebaum, Eliran Kachlon, and Arpita Patra. "Verifiable Relation Sharing and Multi-verifier Zero-Knowledge in Two Rounds: Trading NIZKs with Honest Majority - (Extended Abstract)". In: 2022, pp. 33–56. DOI: 10.1007/978-3-031-15985-5_2.

[Bau+21] Carsten Baum et al. "Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions". In: 2021, pp. 92–122. DOI: 10.1007/978-3-030-84259-8_4.

[Bau+22a] Carsten Baum et al. "Feta: Efficient Threshold Designated-Verifier Zero-Knowledge Proofs". In: 2022, pp. 293–306. DOI: 10.1145/3548606.3559354.

[Bau+22b] Carsten Baum et al. "Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs over $\mathbb{Z}_{2^k}$". In: 2022, pp. 329–358. DOI: 10.1007/978-3-031-15985-5_12.

[Bau+23] Carsten Baum et al. "Publicly Verifiable Zero-Knowledge and Post-Quantum Signatures from VOLE-in-the-Head". In: 2023, pp. 581–615. DOI: 10.1007/978-3-031-38554-4_19.

[Bea95] Donald Beaver. "Precomputing Oblivious Transfer". In: 1995, pp. 97–109. DOI: 10.1007/3-540-44750-4_8.

[Ben+11] Rikke Bendlin et al. "Semi-homomorphic Encryption and Multiparty Computation". In: 2011, pp. 169–188. DOI: 10.1007/978-3-642-20465-4_11.

[Boe+21] Fabian Boemer et al. *Intel HEXL (release 1.2)*. https://github.com/intel/hexl. 2021.

[Bon+19] Dan Boneh et al. "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs". In: 2019, pp. 67–97. DOI: 10.1007/978-3-030-26954-8_3.

[Boo+23] Jonathan Bootle et al. "A Framework for Practical Anonymous Credentials from Lattices". In: 2023, pp. 384–417. DOI: 10.1007/978-3-031-38545-2_13.

[Boy+18] Elette Boyle et al. "Compressing Vector OLE". In: 2018, pp. 896–912. DOI: 10.1145/3243734.3243868.

[Boy+19a] Elette Boyle et al. "Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation". In: 2019, pp. 291–308. DOI: 10.1145/3319535.3354255.

[Boy+19b] Elette Boyle et al. "Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs". In: 2019, pp. 869–886. DOI: 10.1145/3319535.3363227.

[Boy+20] Elette Boyle et al. "Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs". In: 2020, pp. 244–276. DOI: 10.1007/978-3-030-64840-4_9.

[Boy+21] Elette Boyle et al. "Sublinear GMW-Style Compiler for MPC with Preprocessing". In: 2021, pp. 457–485. DOI: 10.1007/978-3-030-84245-1_16.

[BS+14] Eli Ben Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[Bün+20] Benedikt Bünz et al. "Zether: Towards Privacy in a Smart Contract World". In: 2020, pp. 423–443. DOI: 10.1007/978-3-030-51280-4_23.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 259–282. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs.

[Chi+23] Alessandro Chiesa et al. "Eos: Efficient Private Delegation of zkSNARK Provers". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6453–6469. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/chiesa.

[Cra+18] Ronald Cramer et al. "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority". In: 2018, pp. 769–798. DOI: 10.1007/978-3-319-96881-0_26.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. "Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes". In: 2021, pp. 502–534. DOI: 10.1007/978-3-030-84252-9_17.

[Dam+12] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption". In: 2012, pp. 643–662. DOI: 10.1007/978-3-642-32009-5_38.

[Dam+19] Ivan Damgård et al. “New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning”. In: 2019, pp. 1102–1120. DOI: 10.1109/SP.2019.00078.

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. *Line-Point Zero Knowledge and Its Applications*. Cryptology ePrint Archive, Report 2020/1446. https://eprint.iacr.org/2020/1446. 2020.

[Esc+22] Daniel Escudero et al. “TurboPack: Honest Majority MPC with Constant Online Communication”. In: 2022, pp. 951–964. DOI: 10.1145/3548606.3560633.

[Esc+23] Daniel Escudero et al. “SuperPack: Dishonest Majority MPC with Constant Online Communication”. In: 2023, pp. 220–250. DOI: 10.1007/978-3-031-30617-4_8.

[EXY22] Daniel Escudero, Chaoping Xing, and Chen Yuan. “More Efficient Dishonest Majority Secure Computation over $\mathbb{Z}_{2^k}$ via Galois Rings”. In: 2022, pp. 383–412. DOI: 10.1007/978-3-031-15802-5_14.

[FY92] Matthew K. Franklin and Moti Yung. “Communication Complexity of Secure Computation (Extended Abstract)”. In: 1992, pp. 699–710. DOI: 10.1145/129712.129780.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. “Efficient Multi-party Computation: From Passive to Active Security via Secure SIMD Circuits”. In: 2015, pp. 721–741. DOI: 10.1007/978-3-662-48000-7_35.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[GO07] Jens Groth and Rafail Ostrovsky. “Cryptography in the Multi-string Model”. In: 2007, pp. 323–341. DOI: 10.1007/978-3-540-74143-5_18.

[Gol+23] Alexander Golovnev et al. “Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS”. In: 2023, pp. 193–226. DOI: 10.1007/978-3-031-38545-2_7.

[Goy+21] Vipul Goyal et al. “ATLAS: Efficient and Scalable MPC in the Honest Majority Setting”. In: 2021, pp. 244–274. DOI: 10.1007/978-3-030-84245-1_9.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. “Unconditional Communication-Efficient MPC via Hall’s Marriage Theorem”. In: 2021, pp. 275–304. DOI: 10.1007/978-3-030-84245-1_10.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. “Sharing Transformation and Dishonest Majority MPC with Packed Secret Sharing”. In: 2022, pp. 3–32. DOI: 10.1007/978-3-031-15985-5_1.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. “Guaranteed Output Delivery Comes Free in Honest Majority MPC”. In: 2020, pp. 618–646. DOI: 10.1007/978-3-030-56880-1_22.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: 2016, pp. 830–842. DOI: 10.1145/2976749.2978357.

[Li+23] Xiling Li et al. “ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs”. In: *Proceedings of the VLDB Endowment* 16.8 (2023), pp. 1804–1816.

[Liu+22] Hanlin Liu et al. *The Hardness of LPN over Any Integer Ring and Field for PCG Applications*. Cryptology ePrint Archive, Report 2022/712. https://eprint.iacr.org/2022/712. 2022.

[LXY23] Fuchun Lin, Chaoping Xing, and Yizhou Yao. *More Efficient Zero-Knowledge Protocols over $\mathbb{Z}_{2^k}$ via Galois Rings*. Cryptology ePrint Archive, Report 2023/150. https://eprint.iacr.org/2023/150. 2023.

[Pol+23] Antigoni Polychroniadou et al. “Prime Match: A Privacy-Preserving Inventory Matching System”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 6417–6434. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/polychroniadou.

[Rat+23] Mayank Rathee et al. “ELSA: Secure Aggregation for Federated Learning with Malicious Actors”. In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1961–1979. DOI: 10.1109/SP46215.2023.10179468. URL: https://doi.org/10.1109/SP46215.2023.10179468.

[Roy22] Lawrence Roy. "SoftSpokenOT: Quieter OT Extension from Small-Field Silent VOLE in the Minicrypt Model". In: 2022, pp. 657–687. DOI: 10.1007/978-3-031-15802-5_23.
[RS22] Rahul Rachuri and Peter Scholl. "Le Mans: Dynamic and Fluid MPC for Dishonest Majority". In: 2022, pp. 719–749. DOI: 10.1007/978-3-031-15802-5_25.
[Sch+19] Phillipp Schoppmann et al. "Distributed Vector-OLE: Improved Constructions and Implementation". In: 2019, pp. 1055–1072. DOI: 10.1145/3319535.3363228.
[Wen+21a] Chenkai Weng et al. "Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning". In: 2021, pp. 501–518.
[Wen+21b] Chenkai Weng et al. "Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits". In: 2021, pp. 1074–1091. DOI: 10.1109/SP40001.2021.00056.
[WHV24] R. Wang, C. Hazay, and M. Venkitasubramaniam. "Ligetron: Lightweight Scalable End-to-End Zero-Knowledge Proofs. Post-Quantum ZK-SNARKs on a Browser". In: *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 86–86. DOI: 10.1109/SP54263.2024.00086. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00086.
[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. https://github.com/emp-toolkit. 2016.
[Xin+23] Zhibo Xing et al. *Zero-knowledge Proof Meets Machine Learning in Verifiability: A Survey*. 2023. arXiv: 2310.14848 [cs.LG].
[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: 2022, pp. 299–328. DOI: 10.1007/978-3-031-15985-5_11.
[Yan+21] Kang Yang et al. "QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field". In: 2021, pp. 2986–3001. DOI: 10.1145/3460120.3484556.
[YW22] Kang Yang and Xiao Wang. "Non-interactive Zero-Knowledge Proofs to Multiple Verifiers". In: 2022, pp. 517–546. DOI: 10.1007/978-3-031-22969-5_18.
[Zha+20] Fan Zhang et al. "DECO: Liberating Web Data Using Decentralized Oracles for TLS". In: 2020, pp. 1919–1938. DOI: 10.1145/3372297.3417239.
[Zho+23] Zhelei Zhou et al. "Practical Constructions for Single Input Functionality against a Dishonest Majority". In: *Cryptology ePrint Archive* (2023).

A Technical Overview (Continued)

Running the Recursive Check At this point the verifiers have authenticated sharings $[\![w]\!]$ for every wire w coming from an input or multiplication gate. The parties derive authenticated shares for every wire in the circuit by locally handling addition gates, thanks to the linearity of $[\![\cdot]\!]$. The goal now is for them to check that these values satisfy the correct relations, meaning: if $([\![x_i]\!], [\![y_i]\!])$ are inputs to the i-th multiplication gate, and $[\![z_i]\!]$ is the output, then it should hold that $z_i = x_i \cdot y_i$. Also if $[\![w]\!]$ is a wire from an output gate, then $w = 0$. The check for the output gates is quite standard and consists of the verifiers opening a random linear combination of the output gate wires, so we focus our discussion on the multiplication gates instead. For this, we design an interactive protocol among $\{\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n\}$ that receives as input $\{([\![x_i]\!], [\![y_i]\!], [\![z_i]\!])\}_{i=1}^{|C|}$, where $\mathcal{P}$ knows the underlying secrets, and the verifiers accept if $z_i = x_i y_i$ for $i \in [|C|]$, rejecting otherwise. The protocol requires $O(\log |C|)$ rounds and $O(\log |C|)$ field elements of communication, but it can be made non-interactive via Fiat-Shamir, as we discuss shortly. In its first version the protocol also requires interaction in every round between $\mathcal{P}$ and $\{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$, but we discuss how to remove this shortly.

Our verification protocol can be regarded as an instantiation of the fully-linear IOP paradigm from [Bon+19]. However, instead of relying on the FLIOP language, we take a more pragmatic approach and present a self-contained verification protocol. This is in par to recent works that use custom "recursive checks" [GSZ20; Goy+21; Esc+22; Esc+23]. Consider a functionality $\mathcal{F}_{\mathsf{Coin}}$ that samples uniformly random values to both the prover and all verifiers.[13] First, the parties sample

[13] This functionality is used to sample "challenges", and although we *do need it* for the checks involving the instantiation of $\mathcal{F}_{\mathsf{nVOLE}}$ from $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, we *do not need it* for the recursive correctness check, since we will

uniformly random values $\alpha_1, \ldots, \alpha_{|C|} \in \mathbb{F}$, and instead of checking that $x_i \cdot y_i = z_i$ for every $i \in [|C|]$, they check that $\sum_{i=1}^{|C|} \alpha_i x_i y_i = \sum_{i=1}^{|C|} z_i$, which can be written as an inner product $\boldsymbol{x} \cdot \boldsymbol{y} = z$. If at least one product is incorrect, this inner product will also be incorrect with probability at least $O(1 - 1/|\mathbb{F}|)$.

Let $0 < m < |C|$ be a positive integer, which we refer to as the *compression parameter*, and suppose for simplicity that m divides $|C|$, say $|C| = m \cdot \ell$. First, let us arrange $[\![\boldsymbol{x}]\!]$ as a 2-dimensional array $\{x_{i,j}\}_{i \in [\ell], j \in [m]}$, and similarly for $\boldsymbol{y}$. The idea is to reduce the check of one inner product of dimension $m\ell$, to one inner product of dimension ℓ. There are two steps to this: (1) reduce this task to checking m inner products, each of dimension ℓ, and (2) reduce these inner products to only checking *one* inner product, also of dimension ℓ. For the first step begin by noting that z is (allegedly) equal to $\sum_{j=1}^{m} z_j$, where $z_j = \sum_{i=1}^{\ell} x_{i,j} y_{i,j}$ for $j \in [m]$. These are the m inner products of dimension ℓ we reduce the initial check to, but the only caveat is that the verifiers do not have sharings of the result of each product u_j. To address this, $\mathcal{P}$ simply distributes sharings $\{[\![z_j]\!]\}_{j=1}^{m}$ (following the same approach as when $\mathcal{P}$ distributed the extended witness), and the parties check that $\sum_{j=1}^{m} z_j = z$. If the m dot products $z_j = \sum_{i=1}^{\ell} x_{i,j} y_{i,j}$ for $j \in [m]$ are verified to be correct, then it follows that $\boldsymbol{x} \cdot \boldsymbol{y} = z$. An optimization we use is that, instead of checking that $\sum_{j=1}^{m} z_j = z$ (which would involve a zero-test similar to the one used to verify the correctness of output gates), $\mathcal{P}$ only shares $[\![z_j]\!]$ for $j \in [m-1]$, and the verifiers set $[\![z_m]\!] = [\![z]\!] - \sum_{j=1}^{m-1} [\![z_j]\!]$, which automatically forces this condition.

Now, to turn the m secret-shared dot products $z_j = \sum_{i=1}^{\ell} x_{i,j} y_{i,j}$ for $j \in [m]$ into *one* dot product of dimension ℓ, the parties write $(x_{i,1}, \ldots, x_{i,m})$ for $i \in [\ell]$ as the evaluation $(f_i(1), \ldots, f_i(m))$ of a degree-$(m-1)$ polynomial $f_i(\mathtt{X})$, and similarly $y_{i,j} = g_i(j)$. Let $h(\mathtt{X}) := \sum_{i=1}^{\ell} f_i(\mathtt{X}) g_i(\mathtt{X})$, which is a degree-$2(m-1)$ polynomial that should satisfy $h(j) = z_j$ for $j \in [m]$. Note that the parties can locally compute $[\![f_i(\eta)]\!]$ and $[\![g_i(\eta)]\!]$ for any $\eta \in \mathbb{F}$, since they have authenticated shares of enough evaluations of f_i and g_i. However, for h they only have m evaluations, but the degree of h is $2(m-1)$. To address this $\mathcal{P}$ distributes shares of $(2(m-1)+1) - m = m - 1$ additional evaluations of h, say $([\![h(m+1)]\!], \ldots, [\![h(2m-1)]\!])$. At this point the verifiers call $\mathcal{F}_{\mathsf{Coin}}$ to sample a random challenge $\eta \in \mathbb{F}$, and they compute *locally* $\{[\![f_i(\eta)]\!]\}_{i=1}^{\ell}, \{[\![g_i(\eta)]\!]\}_{i=1}^{\ell}, [\![h(\eta)]\!]$. These values should satisfy $h(\eta) = \sum_{i=1}^{\ell} f_i(\eta) g_i(\eta)$, which is a *single* secret-shared dot product of dimension ℓ. Furthermore, thanks to Schwartz-Zippel Lemma, this dot product can only be valid with probability $O(m/|\mathbb{F}|)$ if at least one of the original dot products is incorrect.

Removing Interaction from $\mathcal{P}$ via Fiat-Shamir The recursive check from above can be interpreted as a direct instantiation of the FLIOPs from [Bon+19]. However, our main distinctive aspect is how we turn this interactive protocol into a non-interactive one. This is crucial for our application: our techniques are suitable for the case in which the number of verifiers is relatively large, so minimizing the amount of communication not only among verifiers but between the prover and the verifiers is very well motivated. Towards this, first note that our interactive proof is "public coin" in the sense that the only interaction needed involves the functionality $\mathcal{F}_{\mathsf{Coin}}$. As usual in interactive proofs, for soundness, it is imperative that the prover cannot guess these challenges before sending the messages in the previous rounds. For this, the Fiat-Shamir transform suggests to use a random oracle output $\mathsf{H}(\cdot)$ of previous messages as the challenge, which prevents the prover from choosing earlier messages based on future challenges.

In our multiverifier setting this idea does not work directly. The main issue is that, in our case, the prover sends *different* messages to different verifiers, so it is unclear what "messages" should be passed through the RO. For example, it cannot be a concatenation of all messages across all verifiers since a given verifier $\mathcal{V}_i$ will be missing the other verifier's messages and hence will not be able to compute the challenge on their own, or it cannot be just a subset of messages since this gives a corrupted prover the ability to freely choose some messages without affecting the challenge, which may break soundness. This complication was also noted in the work of Yang and Wang [YW22], which is similar to ours except it is set in the (suboptimal) honest majority regime. As in [YW22, Section 3.3], let us denote the messages that $\mathcal{P}$ sends *privately* to each verifier $\mathcal{V}_i$ by Msg_i.

ultimately use Fiat-Shamir to sample these challenges. However, we only discuss Fiat-Shamir towards the end of this section, so we describe the functionality here.

In [YW22], $\mathcal{P}$ commits to these messages towards *all* verifiers by broadcasting $\{c_i = \mathsf{H}(\mathsf{Msg}_i)\}_{i=1}^n$, and then each $\mathcal{V}_i$ verifies locally that c_i is indeed the evaluation of Msg_i under H, aborting if this is not the case. If no verifier aborts, then the challenge is set to be $\mathsf{H}'(c_1, \ldots, c_n)$. Note that, even though a corrupted $\mathcal{P}$ has the freedom to fix any c_i for a corrupted $\mathcal{V}_i$ (since corrupted verifiers may not check the correctness of the commitments), $\mathcal{P}$ still needs to compute c_i correctly for every honest verifier $\mathcal{V}_i$, meaning that $\mathcal{P}$ must choose the messages to honest verifiers *before* learning the challenge. In [YW22] this is sufficient: in that work there is a *majority* of honest verifiers, and the potential "errors" introduced by $\mathcal{P}$ are all fully determined by the messages sent to honest verifiers; the ability to change corrupted parties' messages does not give the adversary any advantage.

Unfortunately, this idea does not work in our case. As noted above, the adversary can freely choose the commitments c_i for corrupted $\mathcal{V}_i$'s, so the only "unpredictable" values determining the challenge $\mathsf{H}'(c_1, \ldots, c_n)$ are the c_i's for honest verifiers $\mathcal{V}_i$. However, unlike the honest majority setting, a corrupted $\mathcal{P}$ can set valid commitments to valid messages *in advance*, before committing to the errors, and only set the errors after the challenge has been learned. Let us argue why this is the case. First, note that one type of messages that $\mathcal{P}$ sends are difference shares $[\boldsymbol{\delta}]_{\sigma-1}$. These have low degree $\sigma - 1$, and the underlying "secret" $\boldsymbol{\delta}$ is determined by the $\sigma = n - t$ honest verifiers' shares alone. Hence, in this type of messages, a corrupted $\mathcal{P}$ cannot really cheat: as in [YW22], $\mathcal{P}$ has to commit to the difference $\boldsymbol{\delta}$ in advance, before learning the challenge.

The attack vector lies instead in a different type of messages that $\mathcal{P}$ sends: the seeds. For simplicity in the discussion suppose that the number of multiplications is exactly σ (the packing parameter) so that only one random mask $\boldsymbol{r} \in \mathbb{F}^\sigma$ is needed, and furthermore, suppose that its degree-$(n-1)$ shares $[\boldsymbol{r}]_{n-1} = (u^1, \ldots, u^n)$ are not derived from the seeds that $\mathcal{P}$ sends, but rather $\mathcal{P}$ sends each share u^i directly to $\mathcal{V}_i$. A corrupted $\mathcal{P}$ can *fix* in advance the $n - t$ shares u^i of honest verifiers $\mathcal{V}_i$ to a random value, hence learning the challenge in the process. The issue is that at this point the adversary is not committed to the extended witness! Indeed, as we saw $\mathcal{P}$ can set the u^i's for corrupted $\mathcal{V}_i$'s entirely arbitrarily without modifying the challenge, and, unlike the honest majority setting, changing these shares change the underlying mask $\boldsymbol{r}$ and hence change the committed extended witness $\boldsymbol{x} = \boldsymbol{\delta} + \boldsymbol{r}$. The concrete attack is then to set this $\boldsymbol{r}$ in such a way that the extended witness $\boldsymbol{x}$ passes the check in spite of being an incorrect witness. Crucially, note that the t values u^i the adversary can choose are enough to set $\boldsymbol{r} \in \mathbb{F}^\sigma$ (and hence $\boldsymbol{x}$) to any value of its choice in a 1-1 correspondence.

Our solution to this problem lies in ensuring the prover cannot change the challenge only as a functon of the seeds. To achieve this, before the prover sends the proof but after providing the seeds, we ask the verifiers to send *nonces* to $\mathcal{P}$, which are included as input to the RO for computing the challenge. The fact that these are sampled after $\mathcal{P}$ sends the seeds effectively *commits* the prover to these values, preventing the attack highlighted above. This only adds one extra round from the verifiers to the prover before the beginning of the proof.

B Instantiating Programmable Vector OLE ($\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$) over Rings

In this section we discuss our instantiation of the programmable VOLE functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, defined as Functionality 1. Even though the literature contains many works in the direction of VOLE over fields, much less is known for the case of more general rings, specifically Galois rings. Mozzarella [Bau+22b] provides VOLE constructions for $\mathbb{Z}/p^k\mathbb{Z}$, but it does not easily generalize to Galois rings due to their dependency on the "SPDZ2k trick" [Cra+18], which is naturally described over $\mathbb{Z}/p^k\mathbb{Z}$. Recently, the work of [LXY23] provides several VOLE instantiations over Galois rings by generalizing the ones from Wolverine [Wen+21b]. Unfortunately, we cannot use directly the protocols from [LXY23] to instantiate our functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ since they do not achieve *programmability*, which is the property that enables the sender to provide a succinct seed that is expanded to obtain the vector that is multiplied by the scalar. In our context, this property is crucial to let the prover choose the underlying messages of the VOLE and communicate them to the verifiers *succinctly*.

Programmability is used in both Le Mans [RS22] and Superpack [Esc+23] to "link" VOLE instances to OLE instances used to obtain multiplication triples. In [RS22, Appendix D], the authors provide an instantiation of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ *over fields* (that is, they obtain VOLE with programmability) by building on Wolverine [Wen+21b], as in [LXY23]. Once again, the ideas in [RS22] are insufficient for our use-case since they are restricted to fields. At a high level, our approach to instantiate

$\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ is to start from Wolverine [Wen+21b], using the observations from [RS22] to achieve programmability, and these from [LXY23] to instantiate this over Galois rings. In what follows we describe at a high level how such instantiation of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ over Galois rings would work. This is the protocol we use in our proof-of-concept implementation, discussed in Section 5.2. Since most of it follows in a relatively simple manner from existing works, namely [RS22; LXY23; Wen+21b], we keep our discussion rather high level.

Our instantiation of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ in [RS22] follows the same skeleton as [RS22; LXY23; Wen+21b], and is divided into two steps. First, in Section B.1 we instantiate a functionality for chosen-input single-point subfield VOLE, which can be regarded as a general vector OLE functionality where the vector multiplying the scalar has only one non-zero entry that lies in a subfield, which is chosen by one of the parties. Here we can essentially use the instantiation from [LXY23], adding programmability on top, which we discuss in Section B.1. The second part, discussed in Section B.2, consists of using single-point VOLE for instantiating $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. To this end, a few instances of single-point VOLE are used as a "seed" that is then expanded into a proper (programmable) VOLE, where the vector multiplying the scalar is not sparse anymore, but (indistinguishable from) random. This involves applying a PRF that has certain "homomorphic" properties, which is instantiated from the Learning Parity with Noise (LPN) assumption. While [RS22] uses the dual version of the LPN assumption, for efficiency purposes we use the primal version instead, which is done in [LXY23]. Once again, we add programmability on top.

For the rest of the section we let $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ where q is a prime power and $\mathcal{K} = \mathsf{GR}(q, d)$. Moreover, we have been using so far n and t to refer to the number of parties and adversarial threshold respectively, but since this section is only related to two parties, we will "free" these symbols and use them for other purposes in what follows. We will also "free" the symbol k, used so far to denote the exponent of the prime in q. This allows us stick to notation used in previous works, which we believe to be useful for the reader familiar with previous works. For the purpose of this section we will index vectors from 0, and use $\boldsymbol{x}[a, b)$ to denote the (sub)vector $(x_a, \ldots, x_{b-1})$.

B.1 Single-Point Subring VOLE

We begin by defining the chosen-input single-point subring VOLE functionality $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ as Functionality 3. This functionality involves a sender P_A and a receiver P_B, and it allows the sender P_A to obtain $(\boldsymbol{u}, \boldsymbol{w}) \in \mathcal{R}^m \times \mathcal{K}^m$, while P_B obtains $(\Delta, \boldsymbol{v}) \in \mathcal{K} \times \mathcal{R}^m$. This is similar to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ except that here $\boldsymbol{u} \in \mathcal{K}^m$ is a vector with only one non-zero entry, which is an invertible element of $\mathcal{R}$ (not from $\mathcal{K}$).[14] Importantly, P_A can choose $\boldsymbol{u}$, and P_B can choose Δ (hence the "ci"—chosen input—in the functionality notation). This functionality is similar to others in prior works, and we highlight their differences:

- $\mathcal{F}_{\mathsf{spsVOLE}}^{p,d}$ in [Wen+21b, Fig. 6], instantiated by $\Pi_{\mathsf{spsVOLE}}^{p,d}$ in [Wen+21b, Fig. 7]. This is set over fields only and it samples $\boldsymbol{u}$ and Δ internally. In our case these are provided by P_A and P_B respectively.
- $\mathcal{F}_{\mathsf{spVOLE}}^{\mathsf{GR}(2^k,d)}$ in [LXY23, Fig. 17], instantiated by $\Pi_{\mathsf{spVOLE}}^{\mathsf{GR}(2^k,d)}$ in [LXY23, Fig. 10]. This is defined over Galois rings as ours but (1) samples $\boldsymbol{u}$ and Δ internally, and (2) the non-zero entry in $\boldsymbol{u}$ is in $\mathcal{K}$, while in our case it is in $\mathcal{R}^*$. In addition, the authors use their functionality $\mathcal{F}_{\mathsf{spVOLE}}^{\mathsf{GR}(2^k,d)}$ to model the base VOLE needed for the extension, so their functionality contains two types of extend commands: extend, for "small" VOLE queries, and SP-Extend for "long" but sparse (*i.e.* only one non-zero entry) VOLE queries. We model the base VOLE separately, see Section B.1.
- $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ in [RS22, Fig. 20], instantiated by $\Pi_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ in [RS22, Fig. 21]. This is defined over fields while ours is set over Galois rings.

[14] The invertibility requirement is superfluous in the case of finite fields, where every non-zero element is invertible. However, as shown in [LXY23; Liu+22], this turns out to be important later when analyzing the security of LPN in this context.

Functionality 3: $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$

Parameters: Rings $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{K} = \mathsf{GR}(q, d)$, a length n, and party identifiers P_A, P_B.
Initialize: On receiving Init from P_A and (Init, Δ) from P_B, store the global key $\Delta \in \mathcal{K}$, and ignore all subsequent Init commands.
Extend: On receiving $(\mathsf{extend}, n, \alpha \in [n], \beta \in \mathcal{R}^*)$ from P_A and extend from P_B, do:

1. If P_B is honest, sample $\boldsymbol{v} \leftarrow \mathcal{K}^n$. Else, receive $\boldsymbol{v}$ from $\mathcal{A}$.
2. Set $\boldsymbol{u} \in \mathcal{R}^n$ such that $\boldsymbol{u}[i] = 0$ for $i \neq \alpha$, and $\boldsymbol{u}[\alpha] = \beta$. Compute $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathcal{R}^n$.
3. If P_B is corrupt, receive a set $I \subseteq [0, n)$ from $\mathcal{A}$. If $\alpha \in I$, send success to P_B and continue. Else, send abort to both parties, output α to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

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

As we saw above, none of the instantiations in [Wen+21b; RS22; LXY23] are sufficient for our purposes. However, they can be easily adapted. We choose to start from protocol $\Pi_{\mathsf{spVOLE}}^{\mathsf{GR}(2^k,d)}$ in [LXY23, Fig. 10], which almost instantiates our functionality $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$, except it does not allow (honest) P_A and P_B to choose $\boldsymbol{u}$ and Δ respectively, and it does not guarantee that $\boldsymbol{u} \in \mathcal{R}^m$. Fortunately, this is easy to address.

Base VOLE All of the instantiations above follow the same template to obtain single-point VOLE: start from a short VOLE instance to set the VOLE correlation at the non-zero entry, and use a GGM-based puncturable PRF to extend this correlation to cover the zero entries as well. Here, we discuss the base VOLE functionality $\mathcal{F}_{\mathsf{sVOLE}}$ we will use, which is presented below as Functionality 4.

Functionality 4: $\mathcal{F}_{\mathsf{sVOLE}}$

Parameters: Rings $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{K} = \mathsf{GR}(q, d)$, length m, and party identifiers P_A, P_B.
Initialize: On receiving Init from P_A, and (Init, Δ) from P_B, store the global key $\Delta \in \mathcal{K}$, and ignore all subsequent Init commands.
Base VOLE: This procedure can be run multiple times. On receiving (base, ℓ) from P_A and P_B, do:

1. If P_B is honest, sample $\boldsymbol{y} \leftarrow \mathcal{K}^\ell$. Otherwise, receive $\boldsymbol{y} \leftarrow \mathcal{K}^\ell$ from $\mathcal{A}$
2. If P_A is honest, sample $\boldsymbol{x} \leftarrow \mathcal{R}^\ell$ and compute $\boldsymbol{w} := \boldsymbol{y} + \Delta \cdot \boldsymbol{x} \in \mathcal{K}^\ell$. Otherwise receive $\boldsymbol{x} \in \mathcal{R}^\ell$ and $\boldsymbol{w} \in \mathcal{K}^\ell$ from $\mathcal{A}$, and then recompute $\boldsymbol{y} = \boldsymbol{w} - \Delta \cdot \boldsymbol{x} \in \mathcal{K}^\ell$
3. Send $(\boldsymbol{x}, \boldsymbol{w})$ to P_A and $\boldsymbol{y}$ to P_B.

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

$\mathcal{F}_{\mathsf{sVOLE}}$ is a minor variant of the corresponding base VOLE functionality considered in each of the three previous works:

- Functionality $\mathcal{F}_{\mathsf{sVOLE}}^{p,d}$ in [Wen+21b, Fig. 1]. In our case, (1) we consider Galois rings while theirs is only for fields, and (2) we allow P_B to choose Δ instead of having it sampled by the functionality. Also, they keep a dictionary and instead of a base call, they have an extend call, since this functionality is not only used to get the base OLE correlations, but it is also the one they ultimately instantiate (see [Wen+21b, Thm. 4]). In our case we have a functionality $\mathcal{F}_{\mathsf{sVOLE}}$ for the base OLE, and a separate $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ that models VOLE extension.
- Functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{GR}(2^k,d)}$ in [LXY23, Fig. 1]. In our case, (1) we allow P_B to choose Δ instead of having it sampled by the functionality, and (2) we let the vector $\boldsymbol{x}$ sampled by the functionality be in $\mathcal{R}^\ell$ rather than $\mathcal{K}^\ell$ (hence the "s"—which stands for subring—in our functionality name $\mathcal{F}_{\mathsf{sVOLE}}$). As in [Wen+21b] the authors also keep a dictionary and use a extend call.
- Functionality $\mathcal{F}_{\mathsf{sVOLE}}$ in [RS22, Fig. 19]. In our case, we consider Galois rings while theirs is only for fields. They also use a dictionary and an extend command, although they do not use it as

in [Wen+21b; LXY23] since they only use $\mathcal{F}_{\mathsf{sVOLE}}$ for the base VOLE, and not for the extension. We believe our approach of calling this "base" to be clearer.

Note that $\mathcal{F}_{\mathsf{sVOLE}}$ will only be called for short lengths, with the "heavy lifting" being performed by our instantiation $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$; hence, efficiency is not a major concern. We can instantiate $\mathcal{F}_{\mathsf{sVOLE}}$ in a variety of ways:

- From $(N-1)$-out-of-N OT, as in [LXY23, Section 4.1], which extends SoftSpokenOT [Roy22] to Galois rings.
- Extending the field-based VOLE from [Wen+21b, Fig. 5]. This uses correlated oblivious product evaluation with errors (COPEe) as a building block, introduced and constructed from oblivious transfer in [KOS16]. Fortunately, such primitive has also been defined and instantiated over Galois rings in [EXY22].[15]

Remark 6. In the functionality $\mathcal{F}^{\mathsf{GR}(2^k,d)}_{\mathsf{sfVOLE}}$ from [LXY23, Fig. 15], the authors consider a modified version of $\mathcal{F}^{\mathsf{GR}(2^k,d)}_{\mathsf{VOLE}}$ ([LXY23, Fig. 1]) that allows the key Δ to be in $\mathbb{F}_{p^d}$ instead of $\mathcal{K} = \mathsf{GR}(p^k, d)$, which optimizes computation. We can also apply such optimization in our work.

Instantiating single-point VOLE To instantiate $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, we take the protocol $\Pi^{\mathsf{GR}(2^k,d)}_{\mathsf{spVOLE}}$ in [LXY23, Fig. 10] as a starting point. We will make use of $\mathcal{F}_{\mathsf{sVOLE}}$, and also a standard OT functionality $\mathcal{F}_{\mathsf{OT}}$, and a functionality to check equality on values held by the two parties $\mathcal{F}_{\mathsf{EQ}}$. See [Wen+21b; LXY23; RS22] for details.

We obtain our protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ that instantiates $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in the $(\mathcal{F}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model, by performing the following modifications to $\Pi^{\mathsf{GR}(2^k,d)}_{\mathsf{spVOLE}}$ ([LXY23, Fig. 10]):

1. Replace the "Extend" calls to $\mathcal{F}^{\mathsf{GR}(2^k,d)}_{\mathsf{VOLE}}$ ([LXY23, Fig. 1]) by base calls to $\mathcal{F}_{\mathsf{sVOLE}}$.
2. In the initialization phase, P_A sends Init to $\mathcal{F}_{\mathsf{sVOLE}}$ but P_B sends (Init, Δ) instead of just Init.

This does not change the intrinsics of the protocol, and it is straightforward to adapt the proof of Theorem 5 in [LXY23] to show that our resuting protocol, which we denote by $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, is secure.

B.2 Programmable VOLE from Single-Point Subring VOLE

Having an instantiation of $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ at hand, we can proceed to discussing how to use it in order to instantiate $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$. As in previous works [LXY23; RS22; Wen+21b], this is the "VOLE-extension" step that extends a small vector OLE correlation given by $\mathcal{F}_{\mathsf{sVOLE}}$, together with several "sparse" instances given by $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, and obtains a long "dense" (programmable) VOLE instance, and this is instantiated using the LPN assumption. Here, we do not discuss LPN, nor prove the security of our protocol under this assumption, since it will follow from previous works. For a thorough discussion on LPN over rings we refer the reader to [LXY23, Section B], or [Liu+22].[16]

For our exposition, it suffices to know that LPN involves a uniformly sampled matrix $A \in \mathcal{K}^{k\times n}$, where we think of $k \ll n$. Note that the only programmable VOLE extension protocol (over fields) [RS22] uses the dual version of the LPN problem, same as [Boy+18; Boy+19a; CRR21]. This leads to less communication than the primal version at the expense of increased computation times. However, it is known that in practice this trade-off does not play well, with constructions based on primal-LPN offering the best concrete efficiency, even if they have slightly higher communication [Sch+19]. Due to this, instead of following the description from [RS22], we stay closer to works such as [Wen+21b; Sch+19] that make use of the primal-LPN assumption. To

[15] Technically, [Wen+21b] modifies the COPEe in [KOS16] to achieve the "subfield" property required there. Along the same lines, we would need to modify [EXY22] to include the "subring" property. This is straightforward and is approached in a similar way as in [RS22].

[16] Note that we take the matrix A to be defined over $\mathcal{R}$, while in [LXY23] it is taken over the Galois ring extension $\mathcal{K}$. This is because the authors in [LXY23] were not concerned with *subring* VOLE, which we are here. This does not affect security: $\mathcal{K} = \mathsf{GR}(q, d)$ is an $\mathcal{R}$-module or rank d, so an LPN instance where the matrix is in $\mathcal{R}$ and the other elements are in $\mathcal{K}$ corresponds to d instances over $\mathcal{R}$.

the best of our knowledge, our protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ is the first explicit description of a *programmable* VOLE protocol based on *primal* LPN over fields (let alone over Galois rings), which is more computationally efficient than the dual LPN-based construction from [RS22].

We describe our protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ as Protocol 4 below. $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ can be seen as a modified version of the following protocols:

- $\Pi^{p,d}_{\mathsf{sVOLE}}$ in [Wen+21b, Fig. 8], which is restricted to fields and is not programmable.
- $\Pi^{\mathsf{GR}(2^k,d)}_{\mathsf{VOLE}}$ in [LXY23, Fig. 11], which uses Galois rings but is not programmable and does not take the factor multiplying Δ to be in $\mathcal{R}$ as in our case, but in $\mathcal{K}$. Also, the authors only call $\Pi^{\mathsf{GR}(2^k,d)}_{\mathsf{spVOLE}}$ ([LXY23, Fig. 10]), while we, as in [Wen+21b], need to call both $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and $\mathcal{F}_{\mathsf{sVOLE}}$. This is because, as we have mentioned, in [LXY23] the base VOLE is incoporated in the single-point VOLE functionality, while in our case these are separate functionalities.
- $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ in [RS22, Fig. 23], which is programmable but is restricted to fields.

Protocol 4: $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$

Parameters: Rings $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{K} = \mathsf{GR}(q,d)$. Fix n, k, t, and define $\ell = n - k$ and $m = n/t$, where we assume $t \mid n$. Consider a matrix $A \in \mathcal{R}^{k\times n}$ used for the primal LPN assumption, and also parties identifiers P_A, P_B.
Initialize: The parties do the following:

1. On input (Init), P_A sends Init to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and $\mathcal{F}_{\mathsf{sVOLE}}$, and on input (Init, Δ), P_B sends (Init, Δ) to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and $\mathcal{F}_{\mathsf{sVOLE}}$.
2. P_A and P_B send (base, k) to $\mathcal{F}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z}) \in \mathcal{R}^k \times \mathcal{K}^k$ to P_A and $\boldsymbol{y} \in \mathcal{K}^k$ to P_B such that $\boldsymbol{z} = \boldsymbol{y} + \Delta \cdot \boldsymbol{x}$.

Extend: P_A on input $(\mathsf{extend}, \mathsf{seed})$, and P_B on input extend, do the following:

1. P_A parses seed as $(\beta_1, \ldots, \beta_t) \in (\mathcal{R}^*)^t$, samples random $(\alpha_1, \ldots, \alpha_t) \in [m]^t$, and sends $(\mathsf{extend}, m, \alpha_i, \beta_i)$ for $i \in [t]$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, receiving $\boldsymbol{e}_i, \boldsymbol{c}_i) \in \mathcal{R}^m \times \mathcal{K}^m$, where $\boldsymbol{e}_i[\alpha_i] = \beta_i$, and all the other entries are zero.
2. P_B sends extend to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ t times, getting $\boldsymbol{b}_i \in \mathcal{K}^m$ such that $\boldsymbol{c}_i = \boldsymbol{b}_i + \Delta \cdot \boldsymbol{e}_i$ for $i \in [t]$. If either party receives abort from $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in any of these executions, they abort.
3. Let $\boldsymbol{e} = (\boldsymbol{e}_1 \| \cdots \| \boldsymbol{e}_t) \in \mathcal{R}^n$, $\boldsymbol{c} = (\boldsymbol{c}_1 \| \cdots \| \boldsymbol{c}_t) \in \mathcal{K}^n$ and $\boldsymbol{b} = (\boldsymbol{b}_1 \| \cdots \| \boldsymbol{b}_t) \in \mathcal{K}^n$.
   - P_A locally computes $\boldsymbol{u}' = \boldsymbol{x} \cdot A + \boldsymbol{e} \in \mathcal{R}^n$ and $\boldsymbol{w}' = \boldsymbol{z} \cdot A + \boldsymbol{c} \in \mathcal{K}^m$
   - P_B locally computes $\boldsymbol{v}' = \boldsymbol{y} \cdot A + \boldsymbol{b} \in \mathcal{K}^m$.
4. P_A updates $\boldsymbol{x} \leftarrow \boldsymbol{u}'[0:k)$ and $\boldsymbol{z} \leftarrow \boldsymbol{w}'[0:k)$, and P_B updates $\boldsymbol{v}' \leftarrow \boldsymbol{y}[0:k)$, storing these for future extend calls.
5. P_A outputs $\boldsymbol{u} = \boldsymbol{u}'[k:n)$[a] and $\boldsymbol{w} = \boldsymbol{w}'[k:n)$, while P_B outputs $\boldsymbol{v} = \boldsymbol{v}'[k:n)$.

[a] Note that this defines what the function Expand looks like: it is parameterized by the matrix $A \in \mathcal{R}^{k\times n}$, it takes a seed $\mathsf{seed} \in (\mathcal{R}^*)^t$, derives $\boldsymbol{e}$ from it returns $\boldsymbol{u} = (\boldsymbol{x} \cdot A + \boldsymbol{e})[k:n)$

The differences between $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ and $\Pi^{\mathsf{GR}(2^k,d)}_{\mathsf{VOLE}}$ in [LXY23, Fig. 11] are mostly cosmetic, and security of $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ is proven in the exact same way as [LXY23, Thm. 6].

C Proofs for Theorem 1 and Theorem 2

C.1 Proof of Theorem 1

Proof. Let $\mathcal{V}_{\mathcal{A}}$ (resp. $\mathcal{V}_{\mathcal{H}}$) be the set of corrupted verifiers (resp. honest verifiers). We have $|\mathcal{V}_{\mathcal{A}}| \le t$ and $|\mathcal{V}_{\mathcal{H}}| \ge n - t$. We construct a PPT simulator $\mathcal{S}$ that runs $\mathcal{A}$ and is given access to $\mathcal{F}_{\mathsf{nVOLE}}$.
Initialize: For $i \in \mathcal{V}_{\mathcal{H}}$ and $j \in \mathcal{V}_{\mathcal{A}}$, $\mathcal{S}$ simulates $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ and receives a $\Delta^j_i \in \mathcal{K}$ from $\mathcal{V}_j$. Let $\mathcal{V}_{i^*} \in \mathcal{V}_{\mathcal{H}}$ be the honest verifier with the smallest index. $\mathcal{S}$ forwards $\Delta^j_{i^*}$ to $\mathcal{F}_{\mathsf{nVOLE}}$ with the init command.

If $\mathcal{P}$ is honest, $\mathcal{S}$ receives Expand from $\mathcal{F}_{\mathsf{nVOLE}}$ and sends it to all verifiers on behalf of $\mathcal{P}$. Then $\mathcal{S}$ follows the rest of steps to check the consistency of Expand. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ honestly follows the protocol. If all honest verifiers do not receive the same Expand but the check passes, $\mathcal{S}$ aborts.
Extend:

1. If $\mathcal{P}$ is corrupted, for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ receives seed^i from $\mathcal{P}$ and sets $\overline{\mathsf{seed}}^i = \mathsf{seed}^i$. If $\mathcal{P}$ is honest, $\mathcal{S}$ receives $\{\mathsf{seed}^i\}_{i\in\mathcal{V}_\mathcal{A}}$ from $\mathcal{F}_{\mathsf{nVOLE}}$ and sends them to corrupted verifiers.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ for each pair of verifiers $(\mathcal{V}_i, \mathcal{V}_j)$ if exactly one of them is in $\mathcal{V}_\mathcal{A}$.
   - Case 1: $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$. $\mathcal{S}$ receives seed_j^i from $\mathcal{V}_i$ and $\boldsymbol{w}_j^i \in \mathcal{K}^{m+1}$ from $\mathcal{A}$. Then $\mathcal{S}$ replies $(\mathsf{Expand}(\mathsf{seed}_j^i), \boldsymbol{w}_j^i)$ to $\mathcal{V}_i$. For the global key query made by $\mathcal{V}_i$, $\mathcal{S}$ always returns abort to both verifiers and abort.
   - Case 2: $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$. $\mathcal{S}$ receives $\boldsymbol{v}_i^j \in \mathcal{K}^{m+1}$ from $\mathcal{A}$ and replies $\boldsymbol{v}_i^j \in \mathcal{K}^{m+1}$ to $\mathcal{V}_j$. For the guess of the seed, if $\mathcal{P}$ is honest, $\mathcal{S}$ always returns abort, outputs a random value as the seed to $\mathcal{V}_j$, and aborts. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ learns seed^i and follows the functionality honestly.

   For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$, $\mathcal{S}$ sets $\overline{\mathsf{seed}}^i$ to be $\mathsf{seed}_{i^\star}^i$, where recall that $\mathcal{V}_{i^\star} \in \mathcal{V}_\mathcal{H}$ is the honest verifier with the smallest index. Then $\mathcal{S}$ sends $\{\overline{\mathsf{seed}}^i\}_{i\in\mathcal{V}_\mathcal{A}}$ to $\mathcal{F}_{\mathsf{nVOLE}}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also sends $\{\overline{\mathsf{seed}}^i\}_{i\in\mathcal{V}_\mathcal{H}}$ to $\mathcal{F}_{\mathsf{nVOLE}}$.

Consistency Check:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ and sends a uniformly random value $\chi \in \mathcal{K}$ to corrupted verifiers.
2. For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ acts as $\mathcal{V}_i$ to send random values to corrupted verifiers as their shares of $\langle b_i\rangle$. $\mathcal{S}$ also receives the shares of $\langle b_i\rangle$ of honest verifiers from each $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$.
3. $\mathcal{S}$ computes $\hat{u}^i$ for each honest verifier as follows.
   - If $\mathcal{P}$ is corrupted, then $\mathcal{S}$ has received seed^i for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$. For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ computes $u^i = \sum_{\ell=1}^m \chi^\ell \cdot u_\ell^i + u_{m+1}^i$. Then $\mathcal{S}$ samples a random additive sharing $\langle b_i\rangle$ of 0 based on the shares of corrupted verifiers (which have been sent to corrupted verifiers). Then compute $\hat{u}^i = u^i + \sum_{j=1}^n b_j^i$.
   - If $\mathcal{P}$ is honest, $\mathcal{S}$ samples uniform $\hat{u}^i$ for all $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$.

   $\mathcal{S}$ sends $\hat{u}^i$ to corrupted verifiers for all $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$. Then $\mathcal{S}$ receives $\hat{u}^i$ from $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$ and computes $\hat{u} = \sum_{i=1}^n \hat{u}^i$.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ and records $(\tilde{u}^i, \{\tilde{Z}_j^i\}_{j\neq i}, \tilde{Z}_i^i)$ for $i \in \mathcal{V}_\mathcal{A}$. Then $\mathcal{S}$ prepares the messages committed by honest verifiers.
   - When $\mathcal{P}$ is corrupted, $\mathcal{S}$ has computed u^i for $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$. When $\mathcal{P}$ is honest, for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ samples a random additive sharing $\langle b_i\rangle$ of 0 based on the shares of corrupted verifiers (which have been sent to corrupted verifiers). Then compute $u^i = \hat{u}^i - \sum_{j=1}^n b_j^i$.
   - For each $\mathcal{V}_i, \mathcal{V}_j \in \mathcal{V}_\mathcal{H}$ and $i \neq j$, $\mathcal{S}$ samples a random value as $Z_j^i = w_j^i$. For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, $\mathcal{S}$ computes $Z_j^i = w_j^i = u^i \cdot \Delta_i^j + v_i^j$, where $v_i^j = \sum_{\ell=1}^m \chi^\ell \cdot v_{i,\ell}^j + v_{i,m+1}^j$.
   - For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, if $\hat{u} = \sum_{j\in\mathcal{V}_\mathcal{H}} u^j + \sum_{j\in\mathcal{V}_\mathcal{A}} u_i^j$, where $u_i^j = \sum_{\ell=1}^m \chi^\ell \cdot u_{i,\ell}^j + u_{i,m+1}^j$ and

     $$(u_{i,1}^j, \ldots, u_{i,m+1}^j) \leftarrow \mathsf{Expand}(\mathsf{seed}_i^j),$$

     then $\mathcal{S}$ sets $Z_i^i = -\sum_{j\neq i} w_i^j$, where for each $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, $w_i^j = \sum_{\ell=1}^m \chi^\ell \cdot w_{i,\ell}^j + w_{i,m+1}^j$. Otherwise, $\mathcal{S}$ samples a random value as Δ^i and computes $Z_i^i = -\sum_{j\neq i} w_i^j + (\sum_{j\in\mathcal{V}_\mathcal{H}} u^j + \sum_{j\in\mathcal{V}_\mathcal{A}} u_i^j - \hat{u}) \cdot \Delta^i$.
5. $\mathcal{S}$ follows the protocol to check the consistency of all received messages (including the sharing $\langle\hat{u}\rangle$ and all commitments). If all honest verifiers do not receive the same messages but the check passes, $\mathcal{S}$ aborts.
6. $\mathcal{S}$ follows the protocol to open $u^i, \{Z_j^i\}_{j=1}^n$ for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and checks whether $\hat{u} = \sum_{i\in\mathcal{V}_\mathcal{H}} u^i + \sum_{i\in\mathcal{V}_\mathcal{A}} \tilde{u}^i$. If not, $\mathcal{S}$ aborts. Otherwise, for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, if $\tilde{u}^j \neq u_i^j$, $\mathcal{S}$ aborts on behalf of $\mathcal{V}_i$, where $u_i^j = \sum_{\ell=1}^m \chi^\ell \cdot u_{i,\ell}^j + u_{i,m+1}^j$ and $(u_{i,1}^j, \ldots, u_{i,m+1}^j) \leftarrow \mathsf{Expand}(\mathsf{seed}_i^j)$. If $\mathsf{Expand}(\mathsf{seed}_i^j) \neq \mathsf{Expand}(\overline{\mathsf{seed}}^j)$, $\mathcal{S}$ also aborts. $\mathcal{S}$ checks whether $\sum_{i=1}^n Z_j^i = 0$ for all $j \in [n]$. If not, $\mathcal{S}$ aborts. If $\mathcal{P}$ is honest and there exists $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ such that $\Delta_i^j \neq \Delta_{i^\star}^j$, where recall that $\mathcal{V}_{i^\star} \in \mathcal{V}_\mathcal{H}$ is the honest verifier with the smallest index, $\mathcal{S}$ aborts.

Output:

- If $\mathcal{P}$ is corrupted and $\mathcal{S}$ does not abort, for all $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, $\mathcal{S}$ sets $\hat{v}_{i,\ell}^j = u_\ell^i \cdot \Delta_i^j + v_i^j - u_\ell^i \cdot \Delta_{i^\star}^j$ for all $\ell \in [m]$ and sends $\hat{\boldsymbol{v}}_i^j = (\hat{v}_{i,1}^j, \ldots, \hat{v}_{i,m}^j)$ to $\mathcal{F}_{\mathsf{nVOLE}}$. For all $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ sends $\boldsymbol{w}_j^i = (w_{j,1}^i, \ldots, w_{j,m}^i)$ to $\mathcal{F}_{\mathsf{nVOLE}}$. For all $\mathcal{V}_i, \mathcal{V}_j \in \mathcal{V}_\mathcal{A}$ and $i \neq j$, $\mathcal{S}$ sends all-0 vector as $\boldsymbol{v}_i^j$ to $\mathcal{F}_{\mathsf{nVOLE}}$.

– If $\mathcal{P}$ is honest and $\mathcal{S}$ does not abort, for all $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, $\mathcal{S}$ sends $\boldsymbol{v}_i^j = (v_{i,1}^j, \ldots, v_{i,m}^j)$ to $\mathcal{F}_{\mathsf{nVOLE}}$. For all $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ sends $\boldsymbol{w}_j^i = (w_{j,1}^i, \ldots, w_{j,m}^i)$ to $\mathcal{F}_{\mathsf{nVOLE}}$. For all $\mathcal{V}_i, \mathcal{V}_j \in \mathcal{V}_\mathcal{A}$ and $i \neq j$, $\mathcal{S}$ sends all-0 vector as $\boldsymbol{v}_i^j$ to $\mathcal{F}_{\mathsf{nVOLE}}$.

We show the security of Π_{nVOLE} by a series of hybrids.

Hybrid$_0$: This is the same as the real-world execution.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ honestly simulates $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, $\mathcal{F}_{\mathsf{Coin}}$, and $\mathcal{F}_{\mathsf{Commit}}$. The distribution of **Hybrid**$_1$ is identical to that of **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, in the initialization step, if all honest verifiers do not receive the same Expand from $\mathcal{P}$, $\mathcal{S}$ aborts. Assume that H is a collision-resistant hash function, **Hybrid**$_2$ is computationally indistinguishable to **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the response of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ to the global key query and the seed query as described above. For the global key query, when P_B is honest, since Δ is randomly sampled from $\mathcal{K}$, the probability that $\Delta' = \Delta$ is negligible. For the seed query, when P_A is honest and $\mathcal{P}$ is honest, since seed is randomly sampled from S, the probability that $\mathsf{seed} \in I$ is negligible. Thus, **Hybrid**$_3$ and **Hybrid**$_2$ are statistically close.

Hybrid$_4$: In this hybrid, when simulating $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, $\mathcal{S}$ delays the computation of $w_{j,\ell}^i$ when $\mathcal{V}_i$ is honest, and delays the computation of $v_{i,\ell}^j$ when $\mathcal{V}_j$ is honest: After χ are generated, $\mathcal{S}$ generates w_j^i and v_i^j with respect to u_j^i and Δ_i^j. Only at the output step, $\mathcal{S}$ generates $w_{j,\ell}^i$ and $v_{i,\ell}^j$ with respect to $u_{j,\ell}^i$ and Δ_i^j for all $\ell \in [m]$. The distribution of **Hybrid**$_4$ is identical to that of **Hybrid**$_3$.

Hybrid$_5$: In this hybrid, when $\mathcal{P}$ is honest, $\mathcal{S}$ simulates $\mathcal{P}$ by only generating $\{\mathsf{seed}^i\}_{i \in \mathcal{V}_\mathcal{H}}$ at the output step. For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ samples a random value as u^i. Assuming that Expand is a PRG, the distribution of **Hybrid**$_5$ is computationally indistinguishable from that of **Hybrid**$_4$.

Hybrid$_6$: In this hybrid, for each honest verifier, $\mathcal{S}$ simulates $u^i, \hat{u}^i, \langle b_i \rangle$ as described above. The distribution of **Hybrid**$_6$ is identical to that of **Hybrid**$_5$.

Hybrid$_7$: In this hybrid, in Step 6 during the consistency check, if all honest verifiers do not receive the same messages for $\langle \hat{u} \rangle$ and all commitments, $\mathcal{S}$ aborts. Assume that H is a collision-resistant hash function, **Hybrid**$_7$ is computationally indistinguishable to **Hybrid**$_6$.

Hybrid$_8$: In this hybrid, $\mathcal{S}$ simulates Z_i^i for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ as described above. Observe that

$$\begin{aligned} Z_i^i &= (u^i - \hat{u}) \cdot \Delta^i - \sum_{j \neq i} v_j^i \\ &= (u^i - \hat{u}) \cdot \Delta^i - \sum_{j \neq i} (w_i^j - u_i^j \cdot \Delta^i) \\ &= (\sum_{j \in \mathcal{V}_\mathcal{H}} u^j + \sum_{j \in \mathcal{V}_\mathcal{A}} u_i^j - \hat{u}) \cdot \Delta^i - \sum_{j \neq i} w_i^j. \end{aligned}$$

If $\hat{u} = \sum_{j \in \mathcal{V}_\mathcal{H}} u^j + \sum_{j \in \mathcal{V}_\mathcal{A}} u_i^j$, then $Z_i^i = -\sum_{j \neq i} w_i^j$. Otherwise, $Z_i^i = (\sum_{j \in \mathcal{V}_\mathcal{H}} u^j + \sum_{j \in \mathcal{V}_\mathcal{A}} u_i^j - \hat{u}) \cdot \Delta^i - \sum_{j \neq i} w_i^j$. **Hybrid**$_8$ is identical to that of **Hybrid**$_7$.

Hybrid$_9$: In this hybrid, after opening $u^i, \{Z_j^i\}_{j=1}^n$, for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, if $\tilde{u}^j \neq u_i^j$, $\mathcal{S}$ aborts on behalf of $\mathcal{V}_i$. If $\mathsf{Expand}(\mathsf{seed}_i^j) \neq \mathsf{Expand}(\overline{\mathsf{seed}}^j)$, $\mathcal{S}$ aborts. If $\mathcal{P}$ is honest and $\Delta_i^j \neq \Delta_{i^\star}^j$ for some $i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ also aborts. We argue that **Hybrid**$_9$ and **Hybrid**$_8$ are statistically close. It is sufficient to show that when $\mathcal{S}$ aborts in **Hybrid**$_9$, with overwhelming probability, $\mathcal{S}$ also aborts in **Hybrid**$_8$.

– When $\tilde{u}^j \neq u_i^j$, since Δ^i is random and unknown to corrupted verifiers, with overwhelming probability, $\tilde{u}^j \cdot \Delta^i + v_j^i \neq \tilde{Z}_i^j$. Therefore, in this case, $\mathcal{S}$ aborts in **Hybrid**$_8$ with overwhelming probability.
– When $\mathsf{Expand}(\mathsf{seed}_i^j) \neq \mathsf{Expand}(\overline{\mathsf{seed}}^j)$ where recall that $\overline{\mathsf{seed}}^j = \mathsf{seed}_{i^\star}^j$, since χ is uniformly random, with overwhelming probability $u_i^j \neq u_{i^\star}^j$. Then either $u_i^j \neq \tilde{u}^j$ or $u_{i^\star}^j \neq \tilde{u}^j$. In this case, $\mathcal{S}$ aborts in **Hybrid**$_8$ with overwhelming probability as well.

– Given that $\mathcal{S}$ does not abort in the above two cases, we have $\mathsf{Expand}(\mathsf{seed}_i^j) = \mathsf{Expand}(\overline{\mathsf{seed}}^j)$ for all $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$ and $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, and $\hat{u} = \sum_{i=1}^n u^i$. When $\mathcal{P}$ is honest and $\Delta_i^j \neq \Delta_{i^\star}^j$, we have

$$\begin{aligned}\sum_{\ell=1}^{n} Z_j^\ell &= Z_j^i + Z_j^{i^\star} + \sum_{\ell \neq i, i^\star} Z_j^\ell \\ &= u^i \cdot \Delta_i^j + v_i^j + u^{i^\star} \cdot \Delta_{i^\star}^j + v_{i^\star}^j + \sum_{\ell \neq i, i^\star} Z_j^\ell.\end{aligned}$$

Since u^i and $u^{i^\star}$ are two random values given the constraint that $\sum_{\ell=1}^n u^\ell = \hat{u}$, and these two values are not known to corrupted verifiers, with overwhelming probability $\sum_{\ell=1}^n Z_j^\ell \neq 0$.

In summary, $\mathbf{Hybrid}_9$ and $\mathbf{Hybrid}_8$ are statistically close.

$\mathbf{Hybrid}_{10}$: In this hybrid, $\mathcal{S}$ simulates the output step. The difference is that honest verifiers obtain their outputs from $\mathcal{F}_{\mathsf{nVOLE}}$.

– When $\mathcal{P}$ is corrupted, if $\mathcal{S}$ does not abort, then $\mathsf{Expand}(\mathsf{seed}_i^j) = \mathsf{Expand}(\overline{\mathsf{seed}}^j)$ for all $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$ and $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$. In $\mathbf{Hybrid}_9$, each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ takes $w_{j,\ell}^i = u_\ell^i \cdot \Delta_i^j + v_{i,\ell}^j$ and $v_{j,\ell}^i = w_{i,\ell}^j - u_\ell^j \cdot \Delta^i$ for all $j \neq i$ and $\ell \in [m]$ as output. When $\mathcal{V}_j \in \mathcal{V}_\mathcal{H}$, $\Delta_i^j = \Delta^j$, $v_{j,\ell}^i$ is a random value, and $w_{j,\ell}^i = u_\ell^i \cdot \Delta^j + v_{i,\ell}^j$, which have the same distribution as those generated by $\mathcal{F}_{\mathsf{nVOLE}}$. When $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$, $w_{j,\ell}^i = u_\ell^i \cdot \Delta_i^j + v_{i,\ell}^j = u_\ell^i \cdot \Delta_{i^\star}^j + (u_\ell^i \cdot \Delta_i^j + v_{i,\ell}^j - u_\ell^i \cdot \Delta_{i^\star}^j)$ and $v_{j,\ell}^i = w_{i,\ell}^j - u_\ell^j \cdot \Delta^i$. By providing $u_\ell^i \cdot \Delta_i^j + v_{i,\ell}^j - u_\ell^i \cdot \Delta_{i^\star}^j$ and $w_{i,\ell}^j$ to $\mathcal{F}_{\mathsf{nVOLE}}$, the outputs of $\mathcal{V}_i$ in both hybrids are identical.
– When $\mathcal{P}$ is honest, if $\mathcal{S}$ does not abort, then $\mathsf{Expand}(\mathsf{seed}_i^j) = \mathsf{Expand}(\overline{\mathsf{seed}}^j)$ and $\Delta_i^j = \Delta_{i^\star}^j$ for all $\mathcal{V}_j \in \mathcal{V}_\mathcal{A}$ and $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$. Following the same analysis, the outputs of honest verifiers have the same distribution.

Thus, $\mathbf{Hybrid}_{10}$ and $\mathbf{Hybrid}_9$ are identically distributed. Note that $\mathbf{Hybrid}_{10}$ corresponds to the ideal-world execution.

C.2 Proof of Theorem 2

Proof. Let $\mathcal{V}_\mathcal{A}$ (resp. $\mathcal{V}_\mathcal{H}$) be the set of corrupted verifiers (resp. honest verifiers) among $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$. We have $|\mathcal{V}_\mathcal{A}| \leq t$ and $|\mathcal{V}_\mathcal{H}| \geq n - t = \sigma$. We first consider the case where $\mathcal{P}$ is honest. We construct a PPT simulator $\mathcal{S}$ that runs $\mathcal{A}$ and is given access to $\mathcal{F}_{\mathsf{ZK}}$.

Preprocess: In Π_{Prep}, $\mathcal{S}$ emulates the Functionality $\mathcal{F}_{\mathsf{nVOLE}}$.

1. It receives global key shares $\{\Delta^i\}_{i \in \mathcal{V}_\mathcal{A}}$.
2. It samples a random expansion function Expand and distributes the function to all parties.
3. It samples random seeds $\{\mathsf{seed}^i\}_{i \in \mathcal{V}_\mathcal{A}}$, sends them to $\mathcal{A}$, and receives $\{\overline{\mathsf{seed}}^i\}_{i \in \mathcal{V}_\mathcal{A}}$.
4. For every pair of verifiers $(\mathcal{V}_i, \mathcal{V}_j)$, if $\mathcal{V}_i$ is corrupted, $\mathcal{S}$ receives $\boldsymbol{w}_j^i$. If $\mathcal{V}_j$ is corrupted, $\mathcal{S}$ receives $\boldsymbol{v}_i^j$. Then $\mathcal{S}$ prepares the outputs of corrupted verifiers.

Then $\mathcal{S}$ follows the protocol to compute the shares of corrupted verifiers of each authenticated additive sharing using $\{\overline{\mathsf{seed}}^i\}_{i \in \mathcal{V}_\mathcal{A}}$. $\mathcal{S}$ computes the correct shares of corrupted verifiers of each additive sharing using $\{\mathsf{seed}^i\}_{i \in \mathcal{V}_\mathcal{A}}$ and computes the difference between the actual secret and the correct secret. Note that this difference is that the summation of the differences between the actual share and the correct share of each corrupted verifier. This difference is the additive error caused by corrupted verifiers in $\mathcal{F}_{\mathsf{nVOLE}}$.

Next, for each $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$, $\mathcal{S}$ generates random values as the shares of $[\Delta_j^i|_\ell]_t$ of corrupted verifiers for all j, ℓ. For each $\mathcal{V}_i \in \mathcal{V}_\mathcal{A}$, $\mathcal{S}$ receives the shares of $[\Delta_j^i|_\ell]_t$ of honest verifiers for all j, ℓ.

Finally, $\mathcal{S}$ generates random values as the nonces of honest verifiers and receives nonces from corrupted verifiers.

Distribute circuit wire values: For each group of $\sigma \cdot d$ wires, $\mathcal{S}$ samples uniform $\{\lambda_{i,j}\}_{i \in [\sigma], j \in [d]}$ and computes $[\boldsymbol{D}]_{\sigma-1}$. Then $\mathcal{S}$ distributes the shares to all verifiers.

For each $[\Delta_\ell|_i]_t$, let $[\Delta_\ell^\mathcal{H}|_i]_t = \sum_{j \in \mathcal{H}} [\Delta_\ell^j|_i]_t$ and $[\Delta_\ell^\mathcal{A}|_i]_t = \sum_{j \in \mathcal{A}} [\Delta_\ell^j|_i]_t$. Then $[\Delta_\ell|_i]_t = [\Delta_\ell^\mathcal{H}|_i]_t + [\Delta_\ell^\mathcal{A}|_i]_t$. Note that $\mathcal{S}$ learns the shares of $[\Delta_\ell^\mathcal{H}|_i]_t$ of corrupted verifiers, and learns the shares of $[\Delta_\ell^\mathcal{A}|_i]_t$ of honest verifiers.

– $\mathcal{S}$ computes the shares of $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell^{\mathcal{H}}|_i]_t$ of corrupted verifiers and then computes the shares of $\{\langle \lambda_{i,j} \cdot \Delta_\ell^{\mathcal{H}} \rangle\}_{j\in[d]}$ of corrupted verifiers.
– $\mathcal{S}$ computes the shares of $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell^{\mathcal{A}}|_i]_t$ of honest verifiers and then computes the shares of $\{\langle \lambda_{i,j} \cdot \Delta_\ell^{\mathcal{A}} \rangle\}_{j\in[d]}$ of honest verifiers. Since $\mathcal{S}$ knows $\Delta_\ell^{\mathcal{A}}$, $\mathcal{S}$ computes $\lambda_{i,j} \cdot \Delta_\ell^{\mathcal{A}}$ and arbitrarily sets the shares of corrupted verifiers based on the secret and the shares of honest verifiers for all $j \in [d]$.

$\mathcal{S}$ computes the shares of $\{\langle \lambda_{i,j} \cdot \Delta_\ell \rangle\}_{j\in[d]}$ of corrupted verifiers.

Finally, $\mathcal{S}$ computes the shares of $[\![v_{i,j}]\!]$ of corrupted verifiers following the protocol. $\mathcal{S}$ sets the additive error of $v_{i,j}$ to be the additive error of $r_{i,j}$ computed in the first step.

Procedure for Fiat-Shamir:

1. $\mathcal{S}$ honestly emulates the random oracle H.
2. $\mathcal{S}$ acts as the prover $\mathcal{P}$ to compute com_i for $i \in [n]$ by querying H. It sends $\{\mathsf{com}^i\}_{i\in[n]}$ to all verifiers.
3. $\mathcal{S}$ follows the protocol and computes s.

Verify multiplication: $\mathcal{S}$ follows the protocol and traces the shares of corrupted verifiers and the additive errors of $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!], [\![z]\!]$. Then $\mathcal{S}$ simulates Procedure π_{VerifyIP}.

1. **Simulate the inner product compression**:
   (a) In $\pi_{\mathsf{VerifyIPProc}}$ and $\pi_{\mathsf{VerifyIPFinal}}$, $\mathcal{S}$ acts as a prover to send random differences b_js and c_js to verifiers in $\mathcal{V}_{\mathcal{A}}$.
   (b) $\mathcal{S}$ honestly emulates the random oracle H.
   (c) $\mathcal{S}$ follows the protocol and traces the shares of corrupted verifiers and the additive errors of $[\![a]\!], [\![b]\!], [\![c]\!]$.
   (d) $\mathcal{S}$ randomly samples a, b and computes $c = a \cdot b$. Then $\mathcal{S}$ sends (a, b, c) to all verifiers on behalf of $\mathcal{P}$.
2. **Simulate the revealing process**: Let $\delta_a, \delta_b, \delta_c$ be the additive errors. $\mathcal{S}$ simulates the checking process as follows.
   (a) For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ samples random values as the shares of $\langle \theta_1^i \rangle, \langle \theta_2^i \rangle, \langle \theta_3^i \rangle$ of corrupted verifiers and sends them to corrupted verifiers. For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{A}}$, $\mathcal{S}$ receives the shares of $\langle \theta_1^i \rangle, \langle \theta_2^i \rangle, \langle \theta_3^i \rangle$ of honest verifiers. $\mathcal{S}$ arbitrarily sets the shares of $\langle \theta_1^i \rangle, \langle \theta_2^i \rangle, \langle \theta_3^i \rangle$ of corrupted verifiers based on the secrets $\theta_1^i = \theta_2^i = \theta_3^i = 0$ and the shares of honest verifiers.
   (b) $\mathcal{S}$ computes the shares of $\langle o_1 \rangle, \langle o_2 \rangle, \langle o_3 \rangle$ of corrupted verifiers.
   (c) $\mathcal{S}$ emulates the Functionality $\mathcal{F}_{\mathsf{Commit}}$ and receives $\tilde{o}_1^i, \tilde{o}_2^i, \tilde{o}_3^i$ of corrupted verifiers.
   (d) $\mathcal{S}$ follows the protocol to check the consistency of all received messages. If honest verifiers do not receive the same messages but the check passes, $\mathcal{S}$ aborts.
   (e) If $\delta_a, \delta_b, \delta_c$ are all 0, $\mathcal{S}$ sets $o_1 = o_2 = o_3 = 0$. Otherwise, $\mathcal{S}$ samples a random $\Delta \in \mathcal{K}$ and computes $o_1 = \Delta \cdot \delta_a, o_2 = \Delta \cdot \delta_b, o_3 = \Delta \cdot \delta_c$. Then $\mathcal{S}$ samples the shares of $\langle o_1 \rangle, \langle o_2 \rangle, \langle o_3 \rangle$ of honest verifiers based on the shares of corrupted verifiers computed by $\mathcal{S}$ and the secrets o_1, o_2, o_3.
   (f) $\mathcal{S}$ follows the protocol to open the commitments of honest verifiers to $\{o_1^i, o_2^i, o_3^i\}_{i\in\mathcal{V}_{\mathcal{H}}}$. If $\delta_a, \delta_b, \delta_c$ are not all 0 but the check passes, $\mathcal{S}$ aborts.

We prove that the protocol execution in the real world is indistinguishable from the above simulation by a sequence of hybrids.

Hybrid$_0$: This is the same as the real-world execution.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ as follows. $\mathcal{S}$ delays the generation of the outputs of honest verifiers until they are needed. $\mathcal{S}$ also uses uniformly random shares rather than expanding from seeds. When $\mathcal{P}$ is honest, Expand is randomly sampled from the predefined class of expansion functions. By assumption, Expand is a PRG. Thus, **Hybrid**$_1$ is computationally indistinguishable from **Hybrid**$_0$. Note that after replacing shares of honest verifiers by uniformly random values, the secrets of authenticated additive sharings are uniformly random given the shares of corrupted verifiers. This is because for a degree-$(n-1)$ packed Shamir sharing that stores $\sigma = n - t$ secrets, given the shares of corrupted verifiers, there is a one-to-one map between the secrets and the shares of honest verifiers.

Hybrid$_2$: In this hybrid, for each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ first generates the shares of $[\Delta^i_j|_\ell]_t$ of corrupted verifiers. Then when the shares of honest verifiers are needed, $\mathcal{S}$ generates the shares of honest verifiers accordingly. **Hybrid**$_2$ is identically distributed to **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ computes the additive error to the secret of each authenticated additive sharing by using $\{\overline{\mathsf{seed}}^i\}_{i\in\mathcal{V}_{\mathcal{A}}}$ and $\{\mathsf{seed}^i\}_{i\in\mathcal{V}_{\mathcal{A}}}$. This makes no change to the output distribution.

Hybrid$_4$: In this hybrid, for each group of $\sigma \cdot d$ wires, $\mathcal{S}$ samples uniform $\{\lambda_{i,j}\}_{i\in[\sigma],j\in[d]}$. Then $\mathcal{S}$ computes $r_{i,j} = v_{i,j} - \lambda_{i,j}$. As we have argued in **Hybrid**$_1$, $r_{i,j}$ is uniformly random. The distribution of $r_{i,j}$ remains unchanged. After determining the secrets $\{r_{i,j}\}_{i\in[\sigma],j\in[d]}$, $\mathcal{S}$ can compute the shares of honest verifiers accordingly (due to the one-to-one map). **Hybrid**$_4$ is identically distributed to **Hybrid**$_3$.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted verifiers for each $\langle \lambda_{i,j} \cdot \Delta_\ell \rangle$ as described above. From the description, the computed shares satisfy that together with honest verifiers' shares, they form a valid additive sharing $\langle \lambda_{i,j} \cdot \Delta_\ell \rangle$. This makes no change to the output distribution.

Hybrid$_6$: In this hybrid, $\mathcal{S}$ computes the shares of $[\![v_{i,j}]\!]$ of corrupted verifiers following the protocol. $\mathcal{S}$ also sets the additive error of $v_{i,j}$ to be the additive error of $r_{i,j}$ computed in **Hybrid**$_3$. Then the computed shares satisfy that together with the shares of honest verifiers, they form a valid sharing $[\![v_{i,j} + \delta_{v_{i,j}}]\!]$, where $\delta_{v_{i,j}}$ is the computed additive error. This makes no change to the output distribution.

Hybrid$_7$: In the verification of multiplications, $\mathcal{S}$ traces the shares of corrupted verifiers and the additive errors. In π_{VerifyIP}, when sharing $[\![z_j]\!]$ using $[\![r_j]\!]$, $\mathcal{S}$ first samples a random value as b_j and then computes the secret $r_j = z_j - b_j$. After that, $\mathcal{S}$ computes the shares of honest verifiers following **Hybrid**$_4$. **Hybrid**$_7$ is identically distributed to **Hybrid**$_6$.

Hybrid$_8$: In this hybrid, in $\pi_{\mathsf{VerifyIPFinal}}$, $\mathcal{S}$ first randomly samples a, b, c such that $c = a \cdot b$, and then computes $x_{m+1}, y_{m+1}, z_{m+1}$. **Hybrid**$_8$ is identically distributed to **Hybrid**$_7$.

Hybrid$_9$: In this hybrid, $\mathcal{S}$ simulates the checking process. $\mathcal{S}$ first computes the secret values to be reconstructed. In (3), the shares of corrupted verifiers generated by $\mathcal{S}$ for $\langle \theta^i_1 \rangle, \langle \theta^i_2 \rangle, \langle \theta^i_3 \rangle$ satisfy that together with honest verifiers' shares, they form valid additive sharings of 0. Since at least one additive sharing of 0 is prepared by honest verifiers, $\langle o_1 \rangle, \langle o_2 \rangle, \langle o_3 \rangle$ are random additive sharings of o_1, o_2, o_3. In (5), if honest verifiers do not receive the same messages but the check passes, $\mathcal{S}$ aborts. Since H is a random oracle, this happens with negligible probability.

In (6), we should have $o_1 = \Delta \cdot \delta_a, o_2 = \Delta \cdot \delta_b, o_3 = \Delta \cdot \delta_c$. $\mathcal{S}$ follows the above to simulate the shares of honest verifiers. If $\delta_a, \delta_b, \delta_c$ are not all 0 but the check passes, $\mathcal{S}$ aborts. Note that the MAC keys of honest verifiers are uniformly random and unknown to corrupted verifiers. The probability that $\sum_{i\in\mathcal{V}_{\mathcal{A}}} \tilde{o}^i_j + \sum_{i\in\mathcal{V}_{\mathcal{H}}} o^i_j = 0$ for all $j \in [3]$ is negligible. In summary, **Hybrid**$_9$ is computationally indistinguishable from **Hybrid**$_8$.

Hybrid$_{10}$: In the last hybrid, $\mathcal{S}$ no longer generates the shares of honest verifiers, which are not used in the simulation. This corresponds to the ideal world.

Now we move to the case where $\mathcal{P}$ is corrupted. Since honest verifiers do not have inputs, if $|\mathcal{V}_{\mathcal{A}}| < t$, we may run the first $t - |\mathcal{V}_{\mathcal{A}}|$ honest verifiers following the protocol and view them as corrupted verifiers. In the following, we assume that $|\mathcal{V}_{\mathcal{A}}| = t$.

Preprocess: In Π_{Prep}, $\mathcal{S}$ emulates the Functionality $\mathcal{F}_{\mathsf{nVOLE}}$.

1. It receives global key shares $\{\Delta^i\}_{i\in\mathcal{V}_{\mathcal{A}}}$ and the expansion function Expand. Then it distributes the expansion function to all parties.
2. It receives seeds $\{\mathsf{seed}^i\}_{i\in\mathcal{P}}$ from $\mathcal{A}$.
3. For every pair of verifiers $(\mathcal{V}_i, \mathcal{V}_j)$, if $\mathcal{V}_i$ is corrupted, $\mathcal{S}$ receives $\boldsymbol{w}^i_j$. If $\mathcal{V}_j$ is corrupted, $\mathcal{S}$ receives $\boldsymbol{v}^j_i$. Then $\mathcal{S}$ prepares the outputs of corrupted verifiers.

Then $\mathcal{S}$ follows the protocol to compute the shares of corrupted verifiers of each authenticated additive sharing. $\mathcal{S}$ also computes the shares of honest verifiers of each additive sharing (without the MAC part). Then using the shares of all verifiers, $\mathcal{S}$ computes the secrets of each authenticated additive sharing.

Next, for each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ generates random values as the shares of $[\Delta^i_j|_\ell]_t$ of corrupted verifiers for all j, ℓ. For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{A}}$, $\mathcal{S}$ receives the shares of $[\Delta^i_j|_\ell]_t$ of honest verifiers for all j, ℓ.

Finally, $\mathcal{S}$ generates random values as the nonces of honest verifiers and sends the nonces to $\mathcal{P}$.

Distribute circuit wire values: For each group of $\sigma \cdot d$ wires, $\mathcal{S}$ receives the shares of $[\boldsymbol{D}]_{\sigma-1}$ of honest verifiers, which are exactly σ shares. Then $\mathcal{S}$ computes the whole sharing and the secret $\boldsymbol{D}$. Next, $\mathcal{S}$ computes all wire values shared by $\mathcal{P}$.

For each $[\Delta_\ell|_i]_t$, let $[\Delta_\ell^{\mathcal{H}}|_i]_t = \sum_{j\in\mathcal{H}}[\Delta_\ell^j|_i]_t$ and $[\Delta_\ell^{\mathcal{A}}|_i]_t = \sum_{j\in\mathcal{A}}[\Delta_\ell^j|_i]_t$. Then $[\Delta_\ell|_i]_t = [\Delta_\ell^{\mathcal{H}}|_i]_t + [\Delta_\ell^{\mathcal{A}}|_i]_t$. Note that $\mathcal{S}$ learns the shares of $[\Delta_\ell^{\mathcal{H}}|_i]_t$ of corrupted verifiers, and learns the shares of $[\Delta_\ell^{\mathcal{A}}|_i]_t$ of honest verifiers.

- $\mathcal{S}$ computes the shares of $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell^{\mathcal{H}}|_i]_t$ of corrupted verifiers and then computes the shares of $\{\langle\lambda_{i,j} \cdot \Delta_\ell^{\mathcal{H}}\rangle\}_{j\in[d]}$ of corrupted verifiers.
- $\mathcal{S}$ computes the shares of $[\boldsymbol{D}]_{\sigma-1} \cdot [\Delta_\ell^{\mathcal{A}}|_i]_t$ of honest verifiers and then computes the shares of $\{\langle\lambda_{i,j} \cdot \Delta_\ell^{\mathcal{A}}\rangle\}_{j\in[d]}$ of honest verifiers. Since $\mathcal{S}$ knows $\Delta_\ell^{\mathcal{A}}$, $\mathcal{S}$ computes $\lambda_{i,j} \cdot \Delta_\ell^{\mathcal{A}}$ and arbitrarily sets the shares of corrupted verifiers based on the secret and the shares of honest verifiers for all $j \in [d]$.

$\mathcal{S}$ computes the shares of $\{\langle\lambda_{i,j} \cdot \Delta_\ell\rangle\}_{j\in[d]}$ of corrupted verifiers.

Finally, $\mathcal{S}$ computes the shares of $[\![v_{i,j}]\!]$ of corrupted verifiers following the protocol.

Procedure for Fiat-Shamir:

1. $\mathcal{S}$ honestly emulates the random oracle H.
2. For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ honestly checks the correctness of com_i.
3. $\mathcal{S}$ honestly follows the protocol to check $(\mathsf{com}_1, \ldots, \mathsf{com}_n)$. If not all honest verifiers receive the same values but the check passes, $\mathcal{S}$ aborts.

Verify multiplication: $\mathcal{S}$ follows the protocol and traces the shares of corrupted verifiers. $\mathcal{S}$ also follows the protocol to compute the shares of honest verifiers of each additive sharing (without the MAC part). Then $\mathcal{S}$ simulates the checking process in π_{VerifyIP}.

1. For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ samples random values as the shares of $\langle\theta_1^i\rangle, \langle\theta_2^i\rangle, \langle\theta_3^i\rangle$ of corrupted verifiers and sends them to corrupted verifiers. For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{A}}$, $\mathcal{S}$ receives the shares of $\langle\theta_1^i\rangle, \langle\theta_2^i\rangle, \langle\theta_3^i\rangle$ of honest verifiers. $\mathcal{S}$ arbitrarily sets the shares of $\langle\theta_1^i\rangle, \langle\theta_2^i\rangle, \langle\theta_3^i\rangle$ of corrupted verifiers based on the secrets $\theta_1^i = \theta_2^i = \theta_3^i = 0$ and the shares of honest verifiers.
2. $\mathcal{S}$ computes the shares of $\langle o_1\rangle, \langle o_2\rangle, \langle o_3\rangle$ of corrupted verifiers.
3. $\mathcal{S}$ emulates the Functionality $\mathcal{F}_{\mathsf{Commit}}$ and receives $\tilde{o}_1^i, \tilde{o}_2^i, \tilde{o}_3^i$ of corrupted verifiers.
4. $\mathcal{S}$ follows the protocol to check the consistency of all received messages. If honest verifiers do not receive the same messages but the check passes, $\mathcal{S}$ aborts.
5. Let $\tilde{a}, \tilde{b}, \tilde{c}$ denote the reconstruction results received from $\mathcal{P}$. If $(\tilde{a}, \tilde{b}, \tilde{c}) = (a, b, c)$, $\mathcal{S}$ sets $o_1 = o_2 = o_3 = 0$. Otherwise, $\mathcal{S}$ samples a random $\Delta \in \mathcal{K}$ and computes $o_1 = \Delta \cdot (a - \tilde{a}), o_2 = \Delta \cdot (b - \tilde{b}), o_3 = \Delta \cdot (c - \tilde{c})$. Then $\mathcal{S}$ samples the shares of $\langle o_1\rangle, \langle o_2\rangle, \langle o_3\rangle$ of honest verifiers based on the shares of corrupted verifiers computed by $\mathcal{S}$ and the secrets o_1, o_2, o_3.
6. $\mathcal{S}$ follows the protocol to open the commitments of honest verifiers to $\{o_1^i, o_2^i, o_3^i\}_{i\in\mathcal{V}_{\mathcal{H}}}$. If $(\tilde{a}, \tilde{b}, \tilde{c}) \neq (a, b, c)$ but the check passes, $\mathcal{S}$ aborts.

Checking the Emulation Random Oracles: $\mathcal{S}$ checks the following. We only consider distinct queries in the following. (By construction, for the same query, $\mathcal{S}$ will give the same reply as the actual random oracle.)

- If $\mathcal{S}$ returns the same result for two different queries, $\mathcal{S}$ aborts.
- For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, if $\mathcal{S}$ receives a query $(\mathcal{V}_i, \boldsymbol{s}^i, \mathsf{nonce}_i)$ before sending nonce_i and nonce_i is identical to the one he sends to $\mathcal{P}$, $\mathcal{S}$ aborts.
- For each query in the form of $(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$, if for some later query, $\mathcal{S}$ replies com_i for any i, $\mathcal{S}$ aborts.
- For each query in the form of $(\mathsf{IP}, m\ell, \eta_{\mathsf{prev}}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m-1})$ or $(\mathsf{IP}, m, \eta_{\mathsf{prev}}, \{\hat{b}_j\}_{j=1}^{3}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m})$, if for some later query, $\mathcal{S}$ replies η_{prev}, $\mathcal{S}$ aborts.
- If the wire values computed by $\mathcal{S}$ are invalid, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ computes the witness and sends the witness to $\mathcal{F}_{\mathsf{ZK}}$.

We prove that the protocol execution in the real world is indistinguishable from the above simulation by a sequence of hybrids.

Hybrid$_0$: This is the same as the real-world execution.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ as follows. $\mathcal{S}$ delays the generation of the outputs of honest verifiers until they are needed. **Hybrid**$_1$ is identically distributed to **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, for each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, $\mathcal{S}$ first generates the shares of $[\Delta_j^i|_\ell]_t$ of corrupted verifiers. Then when the shares of honest verifiers are needed, $\mathcal{S}$ generates the shares of honest verifiers accordingly. **Hybrid**$_2$ is identically distributed to **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ computes the additive shares of all verifiers by using $\{\mathsf{seed}^i\}_{i\in\mathcal{P}}$. Then $\mathcal{S}$ computes the secret of each authenticated additive sharing. $\mathcal{S}$ also computes the shares of corrupted verifiers of each authenticated additive sharing (including the MAC part).

Hybrid$_4$: In this hybrid, for each group of $\sigma \cdot d$ wires, $\mathcal{S}$ receives the shares of $[\boldsymbol{D}]_{\sigma-1}$ of honest verifiers. Then $\mathcal{S}$ computes the whole sharing and the secrets $\{\lambda_{i,j}\}_{i\in[\sigma],j\in[d]}$. Next, $\mathcal{S}$ computes all wire values shared by $\mathcal{P}$. Computing these values makes no change to the output distribution.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted verifiers for each $\langle \lambda_{i,j} \cdot \Delta_\ell \rangle$ as described above. From the description, the computed shares satisfy that together with honest verifiers' shares, they form a valid additive sharing $\langle \lambda_{i,j} \cdot \Delta_\ell \rangle$. This makes no change to the output distribution.

Hybrid$_6$: In this hybrid, $\mathcal{S}$ computes the shares of $[\![v_{i,j}]\!]$ of corrupted verifiers following the protocol. This makes no change to the output distribution.

Hybrid$_7$: In this hybrid, $\mathcal{S}$ simulates the procedure for Fiat-Shamir. If all honest verifiers do not receive the same $(\mathsf{com}_1, \ldots, \mathsf{com}_n)$ but the check passes, $\mathcal{S}$ aborts. Since H is a random oracle, this happens with negligible probability. **Hybrid**$_7$ is computationally indistinguishable from **Hybrid**$_6$.

Hybrid$_8$: In this hybrid, $\mathcal{S}$ traces the shares of corrupted verifiers in the verification of multiplications. Then $\mathcal{S}$ computes the secret values to be reconstructed by using the additive shares of all verifiers. $\mathcal{S}$ simulates the checking process as described above. In (4), the shares of corrupted verifiers generated by $\mathcal{S}$ for $\langle \theta_1^i \rangle, \langle \theta_2^i \rangle, \langle \theta_3^i \rangle$ satisfy that together with honest verifiers' shares, they form valid additive sharings of 0. Since at least one additive sharing of 0 is prepared by honest verifiers, $\langle o_1 \rangle, \langle o_2 \rangle, \langle o_3 \rangle$ are random additive sharings of o_1, o_2, o_3. In (5), if honest verifiers do not receive the same messages but the check passes, $\mathcal{S}$ aborts. Since H is a random oracle, this happens with negligible probability.

In (6), we should have $o_1 = \Delta \cdot (a - \tilde{a}), o_2 = \Delta \cdot (b - \tilde{b}), o_3 = \Delta \cdot (c - \tilde{c})$. $\mathcal{S}$ follows the above to simulate the shares of honest verifiers. If $(\tilde{a}, \tilde{b}, \tilde{c}) \neq (a, b, c)$ but the check passes, $\mathcal{S}$ aborts. Note that the MAC keys of honest verifiers are uniformly random and unknown to corrupted verifiers. The probability that $\sum_{i\in\mathcal{V}_{\mathcal{A}}} \tilde{o}_j^i + \sum_{i\in\mathcal{V}_{\mathcal{H}}} o_j^i = 0$ for all $j \in [3]$ is negligible. In summary, **Hybrid**$_8$ is computationally indistinguishable from **Hybrid**$_7$.

Hybrid$_9$: In this hybrid, $\mathcal{S}$ checks the emulation of the random oracle. We only consider distinct queries in the following.

- If $\mathcal{S}$ returns the same result for two different queries, $\mathcal{S}$ aborts.
- For each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, if $\mathcal{S}$ receives a query $(\mathcal{V}_i, \boldsymbol{s}^i, \mathsf{nonce}_i)$ before sending nonce_i and nonce_i is identical to the one he sends to $\mathcal{P}$, $\mathcal{S}$ aborts.
- For each query in the form of $(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$, if for later query, $\mathcal{S}$ replies com_i for any i, $\mathcal{S}$ aborts.
- For each query in the form of $\mathsf{IP}, m\ell, \eta_{\mathsf{prev}}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m-1}$ or $(\mathsf{IP}, m, \eta_{\mathsf{prev}}, \{\hat{b}_j\}_{j=1}^{3}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m})$, if for later query, $\mathcal{S}$ replies η_{prev}, $\mathcal{S}$ aborts.

We argue that **Hybrid**$_9$ is computationally indistinguishable from **Hybrid**$_8$. First, since H is a random oracle and its output length is λ bits. For every two different queries, the probability that H outputs the same result is negligible. Since the number of queries made by $\mathcal{A}$ is bounded by polynomial, the first point happens with negligible probability. Second, since nonce_i is randomly chosen by $\mathcal{S}$, the probability that nonce_i is correctly guessed by $\mathcal{P}$ before $\mathcal{S}$ sending it to $\mathcal{P}$ is negligible. Third, since the output of H is uniformly distributed, the probability that H outputs a particular string is negligible. Since the number of queries in the form of $(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$ made by $\mathcal{A}$ is polynomial, the third point happens with negligible probability. Finally, for the same reason as that for the third point, the last point happens with negligible probability.

Thus, **Hybrid**$_9$ and **Hybrid**$_8$ are computationally indistinguishable.

Hybrid$_{10}$: In this hybrid, if the wire values computed by $\mathcal{S}$ are invalid but the check passes, $\mathcal{S}$ aborts. We argue that **Hybrid**$_{10}$ and **Hybrid**$_9$ are computationally indistinguishable.

- For $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, we say $(\mathcal{V}_i, \boldsymbol{s}^i, \mathsf{nonce}_i)$ is a valid query if nonce_i is the correct nonce provided by $\mathcal{V}_i$.
- We say a query in the form of $(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$ is valid if for all $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, com_i is the answer of some valid query.
- We say a query in the form of $(\mathsf{IP}, m\ell, \eta_{\mathsf{prev}}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m-1})$ or $(\mathsf{IP}, m, \eta_{\mathsf{prev}}, \{\hat{b}_j\}_{j=1}^{3}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m})$ is valid if η_{prev} is the answer of some valid query. In particular, if it is for the i-th compression, then η_{prev} should be the answer of some valid query for the $(i-1)$-th compression. Here, when $i = 1$, η_{prev} should be the answer of some valid query starting with Mult.

Thus, for a valid query, we can trace back to an initial query for each $\mathcal{V}_i \in \mathcal{V}_{\mathcal{H}}$, which contains the shares of $\mathcal{V}_i$ of the differences.

- We say the reply to a valid query in the form of

$$(\mathsf{Mult}, \mathsf{com}_1, \ldots, \mathsf{com}_n)$$

  is bad, if the wire values are invalid but the tuple obtained in Step 4.(a) satisfy the inner-product relation.
- We say the reply to a valid query in the form of

$$(\mathsf{IP}, m\ell, \eta_{\mathsf{prev}}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m-1})$$

  or

$$(\mathsf{IP}, m, \eta_{\mathsf{prev}}, \{\hat{b}_j\}_{j=1}^{3}, \{b_j\}_{j=1}^{m-1}, \{c_j\}_{j=1}^{m})$$

  is bad, if the input authenticated triple in $\pi_{\mathsf{VerifyIPProc}}$ (or $\pi_{\mathsf{VerifyIPFinal}}$) is incorrect but the output authenticated triple is correct.

For the first case, when $\chi_1, \ldots, \chi_{m_1}$ are uniformly random, the obtained tuple is correct with probability at most $1/2^{\kappa}$. When $\chi_1, \ldots, \chi_{m_1}$ are obtained from the PRG, the obtained tuple is correct with negligible probability (or otherwise one can distinguish the output from the PRG and a random string by checking whether the obtained tuple is correct). Thus, the probability of a bad reply is negligible. For the second case, in $\pi_{\mathsf{VerifyIPProc}}$, if the input authenticated triple is incorrect, the number of bad replies is bounded by $2(m-1)$. In $\pi_{\mathsf{VerifyIPFinal}}$, if the input authenticated triple is incorrect, the number of bad replies is bounded by $2m$. Thus, the probability of a bad reply is negligible.

Note that if there are no bad replies, then honest verifiers accept the check if and only if the wire values computed by $\mathcal{S}$ are valid, in which case $\mathcal{S}$ can compute the witness of the common statement. Thus **Hybrid**$_{10}$ is computationally indistinguishable from **Hybrid**$_9$.

Hybrid$_{11}$: In the last hybrid, if $\mathcal{S}$ does not abort, $\mathcal{S}$ computes the witness of the common statement and sends it to $\mathcal{F}_{\mathsf{ZK}}$. This corresponds to the ideal world. According to the argument in **Hybrid**$_{10}$, the witness computed by $\mathcal{S}$ is valid. Thus, **Hybrid**$_{11}$ is identically distributed to **Hybrid**$_{10}$.

D List of Functionalities and Protocols

Functionalities

Protocols