Non-Interactive Zero-Knowledge Proofs to Multiple Verifiers

Kang Yang
State Key Laboratory of Cryptology
yangk@sklc.org

Xiao Wang
Northwestern University
wangxiao@cs.northwestern.edu

Abstract

In this paper, we study zero-knowledge (ZK) proofs for circuit satisfiability that can prove to n verifiers at a time efficiently. The proofs are secure against the collusion of a prover and a subset of t verifiers. We refer to such ZK proofs as multi-verifier zero-knowledge (MVZK) proofs and focus on the case that a majority of verifiers are honest (i.e., $t < n/2$). We construct efficient MVZK protocols in the random oracle model where the prover sends one message to each verifier, while the verifiers only exchange one round of messages. When the threshold of corrupted verifiers $t < n/2$, the prover sends $1/2 + o(1)$ field elements per multiplication gate to every verifier; when $t < n(1/2 - \epsilon)$ for any $0 < \epsilon < 1/2$, we can further reduce the communication to $O(1/n)$ field elements per multiplication gate per verifier. Our MVZK protocols demonstrate particularly high scalability: the proofs are streamable and only require a memory proportional to what is needed to evaluate the circuit in the clear.

1 Introduction

Zero-knowledge (ZK) proofs allow a prover $\mathcal{P}$, who knows a witness w, to convince a verifier $\mathcal{V}$ that $C(w) = 0$ for a circuit C, in the way that $\mathcal{V}$ learns nothing beyond the validity of the statement. One important type of ZK proofs is non-interactive ZK (NIZK), where the prover just needs to send one message to a verifier. This is particularly useful as the prover's message (i.e., the proof) can be reused to convince multiple verifiers. The efficiency of NIZK proofs has been significantly improved in recent years, based on different frameworks (e.g., [IKOS07, GKR08, Gro10, GGPR13, BCS16, BCC+16, BBB+18, WTs+18, BCR+19, BBHR19, Set20, BFS20, ZXZS20, AHIV17, BFH+20] and references therein). Another important type of ZK proofs is designated-verifier ZK (DVZK), where an interactive protocol needs to be executed between the prover and the verifier. Compared to NIZK, DVZK protocols can often achieve a higher efficiency to prove to one verifier and scale to a very large circuit with a small memory. For example, state-of-the-art DVZK proof systems [WYKW21, BMRS21, DIO21, YSWW21, BBMH+21] can prove tens of millions of gates per second with very limited bandwidth. However, such an advantage diminishes when the number of verifiers increases: DVZK protocols require the prover to execute the protocol with every verifier, while an NIZK proof enables all verifiers to verify the proof concurrently after the prover generates and publishes the proof.

In this work, we explore the middle ground between NIZK and DVZK: we study the efficiency of ZK proofs when a prover wants to prove to multiple verifiers (i.e., multi-verifier ZK, MVZK in short). This setting was first studied by Abe, Cramer and Fehr [ACF02]. Specifically, we consider that a prover $\mathcal{P}$ needs to convince n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$, and the adversary can potentially corrupt a subset of t verifiers *and* optionally the prover. More specifically, we focus on the honest-majority setting, meaning that $t < n/2$ verifiers could be corrupted and can *collude* with the prover. Such an MVZK protocol is closely connected to DVZK in which the prover can only prove to a designated set of verifiers who are known ahead of the protocol execution. However, due to the fact that there is a majority of honest verifiers, it turns out the

MVZK protocol can achieve some surprising features, e.g., being *non-interactive* between the prover and the verifiers in the information-theoretic setting.

Because of the involvement of multiple verifiers, there are two types of communications: 1) between the prover and verifiers and 2) between different verifiers. We say that the protocol is a *non-interactive* multi-verifier ZK (NIMVZK) proof if the prover only sends one message to each verifier. We say that the protocol is a *strong* NIMVZK proof if it is an NIMVZK and that there is only one round of communication between verifiers. We allow the verifiers to communicate for one round because without any communication between the verifiers, constructing NIMVZKs appears as difficult as constructing NIZKs.

In the MVZK setting, the known protocol [ACF02] or those that can be implicitly constructed from the known techniques [BBC+19, BGIN20] either are *not* concretely efficient or *only* prove some specific circuits instead of generic circuits. Furthermore, none of the prior work considers how to stream the MVZK proofs, which is a crucial property to prove large-scale circuits.

1.1 Our Contribution

In this paper, we propose streamable NIMVZK protocols on generic circuits with both theoretical insights and practical implication. The protocols work in the honest-majority setting, meaning that the number t of corrupted verifiers is less than $n/2$, where n is the total number of verifiers. Compared to NIZKs, our NIMVZK protocols are much cheaper in terms of computational cost and use significantly less memory. Compared to DVZK, our protocols have three advantages: 1) the computation is still cheaper; 2) we can achieve the strongly non-interactive property; and 3) the communication is lower, especially when the number of verifiers is large. Specifically, our results are summarized as follows:

1. We present an information-theoretic NIMVZK protocol, where the prover sends $1 + o(1)$ field elements per multiplication gate to every verifier in one message (thus non-interactive), and the verifiers interact in both communication and rounds *logarithmic* to the circuit size. We also view the protocol as a stepping stone to introduce the following two strong NIMVZK protocols.

2. Assuming a random oracle (and thus in the computational setting), we construct a strong NIMVZK proof based on Shamir secret sharing, where the verifiers only need to communicate for one round. The prover needs to send $1/2 + o(1)$ field elements per multiplication gate to each verifier and the communication cost between verifiers is still logarithmic.

   The challenge is that the message sent by the prover consists of the shares of every verifier that are private information and thus cannot be revealed. This makes the verifiers have no way to compute the public message that can be used as the input of a random oracle in the Fiat-Shamir transform. We proposed an efficient approach to allow Fiat-Shamir to work across multiple verifiers, i.e., enabling the prover to generate a *small* public message that can be *securely* used in the Fiat-Shamir transform, even if a minority of verifiers collude with the malicious prover.

3. When the corruption threshold is smaller (i.e., $t < n(1/2-\epsilon)$ for any $0 < \epsilon < 1/2$), we use packed secret sharing [FY92] to construct a strong NIMVZK protocol for proving a single generic circuit, which further reduces the communication complexity to $O(1/n)$ field elements per multiplication gate per verifier, while the communication complexity between verifiers is logarithmic to the circuit size. If applying the state-of-art secure multi-party computation (MPC) protocol [GPS21] based on packed secret sharing to design an interactive MVZK protocol, the resulting protocol can achieve the same communication complexity. However, the constant in the O notation is significantly larger than our protocol.

   For a single generic circuit, packed secret sharing has been used in MPC protocols [DIK10, GIP15, GIOZ17, GPS21] in the honest-majority setting, but the overhead is often high due to the constraint how to pack the wire values to realize secure evaluation of the circuit. In the ZK setting, our strong NIMVZK protocol can remove the constraint, and achieve optimality for packing wire values and significantly better efficiency for checking correctness of packed sharings. For example, the state-of-the-art MPC protocol [GPS21] using packed secret sharing incurs a total communication cost of $O(n^5k^2)$ to check the consistency between packed input sharings and output sharings, where k is the number of secrets packed in a single sharing. When being improved in the ZK setting, the total communication cost of our protocol can be reduced to $O(n^2k^2)$.

In summary, we designed concretely efficient MVZK protocols, which provide the attractive properties of NIZK (non-interactivity) and DVZK (memory efficiency and prover-computation efficiency). Although it is not applicable to all settings (due to the assumption of honest-majority verifiers), when it is applicable, the performance improvements to existing protocols are huge.

Streamable property of our NIMVZK. Although the communication complexity between the prover and verifiers is linear to the circuit size, all our protocols as described above are *streamable*, meaning that the prover can generate and send the proof on-the-fly and no party needs to store the whole proof during the protocol execution. As a result, the memory consumption of our protocols is proportional to what is needed to evaluate the statement in the clear. Furthermore, we make our strong NIMVZK proofs streamable in the way that the rounds among verifiers keep *unchanged* (i.e., only one round between verifiers is needed for proving multiple batches of gates).

Asymmetric property of our strong NIMVZK. One surprising feature of our two strong NIMVZK protocols in the computational setting is the *asymmetry* among verifiers. Specifically, among all verifiers, a subset of t verifiers only has a *sublinear* communication complexity: each verifier only receives $O(n + \log |C|)$ field elements from the prover, and only needs to send $O(n)$ field elements to other verifiers, where $|C|$ is the number of multiplication gates in a circuit C. It makes the protocols particularly suitable for the applications where the verifiers are a mix of powerful servers and lower-resource mobile devices.

1.2 Applications

Non-interactive MVZK proofs have the following applications:

1. **Drop-in replacement to NIZK and DVZK.** NIMVZK can be used in normal ZK applications as long as the identifies of the verifiers are known ahead of time and satisfy the security requirement (e.g., a majority of verifiers are honest for our NIMVZK protocols). For example, as described in [CK21], a ZK proof could potentially be used by Apple for auditing their Child Sexual Abuse Material detection protocol. Our NIMVZK protocol can be used when such an auditing needs to be performed to multiple agencies efficiently.

2. **Honest-majority MPC with input predicate check.** In some computational tasks, it is desired to execute the MPC protocol among multiple parties only if the input of every party is valid, where the validity is defined by some predicate. Although generic MPC can realize this functionality, using our NIMVZK protocols could further reduce the overhead of proving the predicate. As our protocols are based on Shamir secret sharing, it can be seamlessly integrated with MPC protocols also based on Shamir secret sharing.

3. **Private aggregation systems.** Systems like Prio [CGB17] use a set of servers to collect and aggregate users' data. To prevent mistakes and attacks, users need to prove to the servers that their data is valid, which was done via *secret-shared non-interactive proof* in Prio. However, the protocol assumes the prover not to collude with any verifier for soundness. Our protocol could be a more efficient alternative and is sound even when a user colludes with a minority of servers. On the other hand, for zero-knowledge,

Prio can tolerate all-but-one corrupted servers, while our protocols need to assume an honest majority of servers.

1.3 Related Work

The concept of MVZK proofs was first discussed by Burmester and Desmedt [BD91], where they focus on how to save broadcasts. Lepinski, Micali and shelat [LMs05] proposed a fair ZK proof, which ensures that even malicious verifiers who collude with the prover can learn nothing beyond the validity of the statement if the honest verifiers accept the proof. More recently, Baldimtsi et al. [BKZZ20] proposed a crowd verifiable zero-knowledge proof, where the focus is to transform a Sigma protocol to their setting. All of the above works focus on extending the ZK functionality rather than the concrete efficiency of ZK proofs.

Abe, Cramer and Fehr [ACF02] first studied the MVZK setting, and proposed a strong NIMVZK protocol for circuit satisfiability if at most $t < n/3$ verifiers are corrupted. Their protocol builds on the technique [Abe99], and adopts the Pedersen commitment and verifiable secret sharing. Due to the usage of public-key operations for every non-linear gate, their protocol is *not* concretely efficient.

Although Boneh et al. [BBC$^+$19] did *not* explicitly consider the MVZK setting, the ZK proofs proposed by them work in this setting. However, these protocols are only efficient applicable for circuits that can be represented by low-degree polynomials (instead of generic circuits). Recently, Boyle et al. [BGIN20] shown how to use the ZK proof [BBC$^+$19] to design honest-majority MPC protocols with malicious security. However, they only considered how to prove correctness of degree-2 relations, and did *not* involve MVZK proofs on generic circuits yet. In addition, Boyle et al. [BGIN20] proposed an approach based on Fiat-Shamir to make the ZK proof on inner-product tuples non-interactive, where the difference between the secret and randomness needs to be sent. One can *generalize* their approach into our MVZK framework, and make the resulting MVZK proof strongly non-interactive. However, their approach requires $3\times$ larger communication than ours. Both works [BBC$^+$19, BGIN20] did *not* consider how to make the ZK proofs *streamable*, which is a crucial property to prove large-scale circuits, and is addressed by our work.

The Prio proof system [CGB17] works in the MVZK setting when assuming the prover *cannot* collude with any verifier, as additive secret sharing is used. In terms of efficiency, Prio needs $2\times$ larger communication cost than our NIMVZK proof, and requires the computation complexity of $O(|C| \log^2 |C|)$ while our proof has the *linear* computation complexity.

One can also use maliciously secure MPC protocols in the honest-majority setting (e.g., [GIP15, LN17, NV18, CGH$^+$18, BGIN20, GS20, GSZ20, GLO$^+$21, GPS21]) to directly obtain interactive MVZK proofs. However, both of communication and computation costs will be significantly larger than our NIMVZK protocols.

2 Preliminaries

We discuss some important preliminaries here and provide more preliminaries (e.g., security model) in Appendix A.

Notation. We use λ and ρ to denote the computational and statistical security parameters, respectively. We use $x \leftarrow S$ to denote that sampling x uniformly at random from a finite set S. For $a, b \in \mathbb{Z}$ with $a \leq b$, we write $[a, b] = \{a, \ldots, b\}$. We will use bold lower-case letters like $\boldsymbol{x}$ for column vectors, and denote by x_i the i-th component of $\boldsymbol{x}$ with x_1 the first entry. For two vectors $\boldsymbol{x}, \boldsymbol{y}$ of dimension m, $\boldsymbol{x} \odot \boldsymbol{y}$ denotes the inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$ (i.e., $\boldsymbol{x} \odot \boldsymbol{y} = \sum_{i \in [1,m]} x_i \cdot y_i$). Sometimes, when the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is 1 (i.e., $\boldsymbol{x} = x$ and $\boldsymbol{y} = y$), we abuse the notation $\boldsymbol{x} \odot \boldsymbol{y}$ to denote the multiplication $x \cdot y$ for the sake of simplicity. We use $\log_k$ to denote the logarithm in base k, and denote by $\log$ the logarithm notation $\log_2$ for simplicity. For a finite field $\mathbb{F}$, we use $\mathbb{K}$ to denote a degree-r extension field of $\mathbb{F}$. In particular, we

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

This functionality runs with a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$. Let $\mathcal{H}$ denote the set of honest verifiers. This functionality operates as follows:

1. Upon receiving $(\mathsf{prove}, C, \boldsymbol{w})$ from $\mathcal{P}$ and (verify, C) from $\mathcal{V}_i$ for all $i \in [1, n]$ where C is a circuitover a field $\mathbb{F}$, set $b := \mathsf{true}$ if $C(\boldsymbol{w}) = 0$ and $b := \mathsf{false}$ otherwise.
2. Send b to the adversary. For each $i \in \mathcal{H}$, wait for an input from the adversary, and then do the following:
   - If it is $\mathsf{continue}_i$, send b to the verifier $\mathcal{V}_i$.
   - If it is abort_i, send abort to the verifier $\mathcal{V}_i$.

Figure 1: **Zero-knowledge functionality for multiple verifiers in the honest-majority setting.**

fix some monic, irreducible polynomial $f(X)$ of degree r and write $\mathbb{K} \cong \mathbb{F}[X]/f(X)$. Every field element $w \in \mathbb{K}$ can be denoted uniquely as $w = \sum_{h\in[1,r]} w_h \cdot X^{h-1}$ with $w_h \in \mathbb{F}$ for all $h \in [1, r]$. When we write arithmetic expressions involving both elements of $\mathbb{F}$ and elements of $\mathbb{K}$, it is understood that field elements in $\mathbb{F}$ are viewed as the polynomials lying in $\mathbb{K}$ that have only constant terms. For a circuit C, we use $|C|$ to denote the number of multiplication gates.

Zero-knowledge functionality. Our ZK functionality for proving circuit satisfiability against multiple verifiers is shown in Figure 1. Let n be the total number of verifiers. We consider the MVZK protocols in the honest-majority setting, i.e, the adversary allows to corrupt at most $t < n/2$ verifiers. The adversary is also allowed to corrupt the prover. When the prover is honest, functionality $\mathcal{F}_{\mathsf{mvzk}}$ defined in Figure 1 captures *zero-knowledge*, meaning that t malicious verifiers cannot learn any information on the witness. When the prover is malicious, $\mathcal{F}_{\mathsf{mvzk}}$ captures *soundness*, i.e., the malicious prover cannot make the honest verifiers accept if $C(\boldsymbol{w}) \neq 0$, even though it *colludes* with t malicious verifiers. From the definition of $\mathcal{F}_{\mathsf{mvzk}}$, it requires that the ZK protocol realizing $\mathcal{F}_{\mathsf{mvzk}}$ supports *knowledge extraction*.

We can consider MVZK protocols as special MPC protocols. Thus, we adopt the notion of *security with abort* in the MPC setting to define $\mathcal{F}_{\mathsf{mvzk}}$ and other functionalities defined in the subsequent sections, where the corrupted verifiers may receive output while the honest verifiers do not. Our definition does not guarantee *unanimous abort*, meaning that some honest verifiers may receive output while other honest verifiers abort. Nevertheless, it is easy to tune our protocols to satisfy the security notion of *unanimous abort*, by having the verifiers broadcast whether they will abort or not at the end of the protocol execution [GL05].

Communication model. The default communication between the prover and verifiers is private channel, unless otherwise specified. We assume that all verifiers are connected via authenticated channels. In the computational setting, the prover sometimes needs to communicate with all verifiers over a broadcast channel. Since we allow abort, the broadcast channel can be established using a standard echo-broadcast protocol [GL05], where the communication overhead can be improved to be constant small using a collision-resistant hash function.

In our strong NIMVZK protocols, the verifiers need to exchange the shares in one round at the end of protocol execution. In parallel with the communication of shares, every verifier can send the hash output of the messages broadcast by the prover to all other verifiers. Therefore, although the echo-broadcast protocol is used in our MVZK proofs, we can still achieve strongly non-interactive.

Linear secret sharing scheme. In our NIMVZK protocols, we will extensively use *linear secret sharing schemes* (LSSSs) with a threshold t. A t-out-of-n LSSS enables a secret x to be shared among n parties, such that no subset of t parties can learn any information on x, while any subset of $t + 1$ parties can reconstruct the secret. To align with the description of our NIMVZK protocols, we let the prover $\mathcal{P}$ play the role of the dealer and let every verifier $\mathcal{V}_i$ obtain the shares. We require that LSSS supports the following procedures:

- $[x] \leftarrow \mathsf{Share}(x)$: In this procedure, a dealer $\mathcal{P}$ shares a secret x among the parties $\mathcal{V}_1, \ldots, \mathcal{V}_n$, such that $\mathcal{V}_i$ gets a share x^i for $i \in [1, n]$. The sharing of x output by this procedure is denoted by $[x]$.

- $x \leftarrow \mathsf{Open}([x])$: Given a sharing $[x]$, this procedure is executed by parties $\mathcal{V}_1, \ldots, \mathcal{V}_n$. At the end of the execution, if $[x]$ is not valid, then all honest parties abort; otherwise, every party will output x.

- *Linear combination*: Given the public coefficients $c_0, c_1, \ldots, c_\ell$ and $[x_1], \ldots, [x_\ell]$, the parties $\mathcal{V}_1, \ldots, \mathcal{V}_n$ can *locally* compute $[y] = \sum_{i=1}^{\ell} c_i \cdot [x_i] + c_0$, such that $y = \sum_{i=1}^{\ell} c_i \cdot x_i + c_0$ holds.

We describe two LSSS instantiations shown in Appendix A, where one is Shamir secret sharing and the other is packed secret sharing (a generalization of Shamir secret sharing). For Shamir secret sharing, for a vector $\boldsymbol{x} = (x_1, \ldots, x_m)$, we will use $[\boldsymbol{x}]$ to denote $([x_1], \ldots, [x_m])$. For packed secret sharing, for a vector $\boldsymbol{x} \in \mathbb{F}^k$, we will use $[\boldsymbol{x}]$ to denote a single packed sharing that stores k secrets of $\boldsymbol{x}$. We assume that the shares of any t parties are uniformly random, which is satisfied by the two instantiations.

3 Technical Overview

We describe the ideas in our NIMVZK protocols and how we come up with these constructions in this section. We leave the full details and their proofs of security in later sections.

3.1 Black-Box Constructions of NIMVZK Proofs

Multi-verifier zero-knowledge proof and its non-interactive version are closely connected to popular concepts in prior work but also with crucial differences. As a warm-up, we discuss how to construct MVZK proofs using well-studied building blocks in a *black-box* way.

Given a linear probabilistically checkable proof (PCP) [BCI+13], one can construct an NIMVZK proof, where the verifiers communicate for 3 rounds between themselves (and thus is not a strong NIMVZK). In detail, the prover can locally construct a PCP proof π and then secretly share π to all verifiers. Now, the verifiers can locally perform all linear queries using a coin-tossing functionality to generate the randomness. Any linear query can be answered easily as long as the secret sharing scheme is linear (so that the queries can be computed efficiently) and verifiable (so that malicious verifiers cannot affect the soundness). The protocol requires coin tossing (at least 2 rounds) followed by reconstructing the secrets on the query results (at least 1 round), and thus at least 3 rounds are needed between verifiers. Similarly, a linear interactive oracle proof (IOP) such as [BBC+19] can be converted to an NIMVZK as well, but the round complexity between verifiers is increased as the rounds of queries increase.

In theory, it is possible to obtain an NIMVZK proof, where the verifiers only interact for 2 rounds (instead of 3 rounds) from linear PCP using a two-round maliciously secure MPC protocol such as [ACGJ19]. Following the above structure and after the prover sends the shares of a proof π, one can instead use a two-round MPC protocol where each party inputs their shares of π and randomness, and use the MPC protocol to emulate the verifier of linear PCP. However, this idea incurs a very high cost among verifiers due to: 1) non-black-box use of the PCP verifier, and 2) the high overhead in the two-round MPC protocol. What's more, the idea does not seem extendable to allow 1-round communication among verifiers (i.e., strong NIMVZK).

3.2 Information-Theoretic Non-Interactive MVZK

We introduce our MVZK proofs starting from an *information-theoretic* NIMVZK protocol. Note that the black-box construction as described above is already information-theoretically secure, but the computation and communication between the prover and verifiers are often high (i.e., the black-box construction is significantly less efficient than our information-theoretic protocol).

Our approach for NIMVZK. Our NIMVZK proofs follow the "commit-and-prove" paradigm, where secrets are committed using Shamir sharings and the security of commitments is guaranteed in the honest-majority setting. At a high level, our information-theoretic protocol (as well as other protocols discussed later) have the following steps.

1. For the output z of each circuit-input gate or multiplication gate, the prover runs $\mathsf{Share}(z)$ to distribute the shares of $[z]$ to all verifiers. Since LSSS is used, the addition gates can be locally computed by the verifiers.

2. For a circuit with N multiplication gates, we have N multiplication triples $([x_i], [y_i], [z_i])$ over a field $\mathbb{F}$ that the verifiers need to check. All parties jointly sample a uniform element $\chi \in \mathbb{K}$, and then compute the inner-product tuple:

$$[\boldsymbol{x}] := \left([x_1], \chi \cdot [x_2], \ldots, \chi^{N-1} \cdot [x_N]\right), \ [\boldsymbol{y}] := ([y_1], [y_2], \ldots, [y_N]), \ [z] := \textstyle\sum_{i\in[1,N]} \chi^{i-1} \cdot [z_i].$$

   If there exists one *incorrect* multiplication triple, then the inner-product tuple defined as above is also *incorrect*, except with probability $\frac{N-1}{|\mathbb{K}|}$.

3. The verifiers check correctness of the inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with logarithmic communication.

In the information-theoretic setting, the verifiers can call a coin-tossing functionality (shown in Appendix A) to sample the coefficient χ, but χ is *not* available to the prover while keeping non-interactive between the prover and verifiers. The distributed ZK proofs by Boneh et al. [BBC+19] could check correctness of an inner-product tuple, but it only works when the prover knows the secrets. To use their protocol directly, we would need the verifiers to send χ to the prover, and then the round complexity between the prover and verifiers will be at least 3 rounds.

Our task is to design a non-interactive protocol that verifies correctness of an inner-product tuple, where the secrets are shared among verifiers and *unknown to the prover*. We adapt the checking approach by Goyal et al. [GS20, GSZ20] (building on the technique [BBC+19]) from the MPC setting to the MVZK setting, and construct a verification protocol to check correctness of inner-product tuples. In particular, our verification protocol makes the prover generate the random sharings and random multiplication triples (instead of letting the verifiers run the DN multiplication protocol [DN07] that is done in [GS20, GSZ20]), which is sufficient for MVZK as zero-knowledge only needs to hold for an honest prover.

3.3 Distributing Fiat-Shamir for Strong Non-Interactive MVZK

With the above preparation, we now discuss how to construct a strong NIMVZK proof, where the verifiers communicate for only one round. This is a highly non-trivial task, as it is even unclear how to sample a random coefficient $\chi \in \mathbb{K}$ as needed in step 2. Since every verifier can only send one message to other verifiers, using a secure coin-tossing protocol is not possible. The other randomness source that we can use is random oracle (i.e., adopting the Fiat-Shamir heuristic). However, only the shares are sent by the prover where the shares need to be kept secret, and thus the verifiers has no way to compute a public message that can be used as the input of a random oracle. This was in fact attempted in the distributed ZK proof [BBC+19] as well, but their non-interactive solution does not allow the prover to collude with any verifier.

Let's first review how Boneh et al. [BBC+19] use Fiat-Shamir in the case that all verifiers do not collude with the prover. Suppose that the prover $\mathcal{P}$ sends a message Msg_i along with a randomness r_i to a verifier $\mathcal{V}_i$ for $i \in [1, n]$, where Msg_i and r_i need to be kept secret. Every verifier $\mathcal{V}_i$ can send $\nu_i := \mathsf{H}(\mathsf{Msg}_i, r_i)$ to other verifiers where H is a random oracle, and then generates a random challenge $\chi := \bigoplus_{i\in[1,n]} \nu_i$, when ignoring some details for simplicity. Prover $\mathcal{P}$ can also compute the challenge χ as it knows all messages and randomness. When the prover is corrupted (and thus we are concerning soundness), all verifiers are assumed to be honest and thus can exchange the correct values $\{\nu_i\}_{i\in[1,n]}$, so that the verifiers can compute a random challenge χ to execute the protocol. However, when $\mathcal{P}$ colludes with a verifier $\mathcal{V}_{i^*}$, this method does not work anymore: $\mathcal{P}$ can cheat when the challenge is some value $\chi* \neq \bigoplus_{i\in[1,n]} \mathsf{H}(\mathsf{Msg}_i, r_i)$; after receiving the values of other verifiers, $\mathcal{V}_{i^*}$ can compute $\nu_{i^*} = \mathsf{H}(\mathsf{Msg}_{i^*}, r_{i^*})$ and the correct challenge χ, and then send $\nu'_{i^*} = \nu_{i^*} \oplus \chi \oplus \chi*$ to every other verifier such that the invalid proof can still go through.

Because of the round-complexity requirement on the verifier side, we cannot let the verifiers to sample χ. So it appears that in order to get a strong NIMVZK, we must find an approach to enable the prover to generate public messages via some sort of Fiat-Shamir transformation in the distributed setting so that: 1) the protocol tolerates the collusion of the prover and a minority of verifiers, and 2) does not require the verifiers to interact more than one round. Let H, H' be two random oracles with exponentially large ranges. Our technique to support Fiat-Shamir is presented as follows.

1. Suppose that the prover $\mathcal{P}$ sends (Msg_i, r_i) to every verifier $\mathcal{V}_i$ over a private channel, where Msg_i consists of the shares held by $\mathcal{V}_i$ for our NIMVZK protocols.

2. Now, $\mathcal{P}$ also broadcasts commitments $com_i := \mathsf{H}(\mathsf{Msg}_i, r_i)$ for all $i \in [1, n]$ to all verifiers, where the broadcast does not increase the rounds between verifiers that has been explained in Section 2.

3. Every verifier $\mathcal{V}_i$ checks that $com_i = \mathsf{H}(\mathsf{Msg}_i, r_i)$. As we assume that $t < n/2$, we can guarantee that a majority of commitments in $com_1, \ldots, com_n$ are computed correctly.

4. The verifiers can generate a random challenge $\chi := \mathsf{H}'(com_1, \ldots, com_n)$, as they now know the public messages $com_1, \ldots, com_n$. Then, the verifiers use χ to transform the verification of N multiplication triples into that of an inner-product tuple as described in Section 3.2.

If H has a 2λ-bit output length and thus is collision-resistant, then it would make com_i binding and the proof can easily go through, where all the commitments $\{com_i\}$ held by $n - t$ honest verifiers will uniquely define the secrets on all wires. However, we make a key observation that it is sufficient to prove security, if the challenge χ is guaranteed to be defined after the secrets on all wires have been determined (i.e., χ is independent of these secrets). Therefore, it is unnecessary to require the collision resistance for H, but rather we only need H to be second preimage-resistant, which allows to achieve better efficiency, e.g., using the construction [DNNR17]. In particular, if χ has been defined and known by the malicious prover, then it must make a query $(com_1, \ldots, com_n)$ to random oracle H'. Then, the malicious prover cheats to find a pair (Msg'_i, r'_i) associated with χ, and then sends it to some honest verifier $\mathcal{V}_i$. The cheat will not be detected only if $com_i = \mathsf{H}(\mathsf{Msg}'_i, r'_i)$, which is equivalent to find either a preimage or a second preimage of com_i.

The Fiat-Shamir approach as described above only introduces a small communication overhead, i.e., $O(n^2\lambda)$ bits in total between the prover and all verifiers that is independent of the circuit size.

Strong NIMVZK for inner-product tuples. Once we enable Fiat-Shamir, we also get another benefit that now the challenge χ is also known to the prover $\mathcal{P}$, and thus we can use the one-round protocol [BBC$^+$19, BGIN20] for verifying inner-product tuples with sublinear communication. We further simplify the technique by Boneh et al. [BBC$^+$19] by avoiding the use of verifiable secret sharing, and also optimize the communication and computation costs. Our strong NIMVZK protocol for inner-product tuples can be viewed as a non-interactive version of our inner-product verification protocol in the information theoretic setting (which in turn is inspired by Goyal et al. [GS20, GSZ20]). This protocol also adapts the technique by Baum et al. [BMRS21] from the DVZK setting to the MVZK setting in order to further improve the computation efficiency. At a high level, our strong NIMVZK protocol for proving correctness of an inner-product tuple works as follows:

1. Suppose that all verifiers hold the shares of an inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is N. The prover $\mathcal{P}$ knows the secrets $(\boldsymbol{x}, \boldsymbol{y}, z)$, and wants to prove $z = \boldsymbol{x} \odot \boldsymbol{y}$.

2. The verifiers *recursively* reduce the dimension of $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to 2. This is done by splitting an inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with dimension m into $([\boldsymbol{a}_1], [\boldsymbol{b}_1], [c_1])$ and $([\boldsymbol{a}_2], [\boldsymbol{b}_2], [c_2])$ with dimension $m/2$ and then using a protocol to compress the two inner-product tuples into one tuple, where $[\boldsymbol{x}] = ([\boldsymbol{a}_1], [\boldsymbol{a}_2])$, $[\boldsymbol{y}] = ([\boldsymbol{b}_1], [\boldsymbol{b}_2])$ and $[z] = [c_1] + [c_2]$.

3. To realize the splitting step, $\mathcal{P}$ can directly distribute the shares of $[c_1] = [\boldsymbol{a}_1 \odot \boldsymbol{b}_1]$ to all verifiers. However, this does not support Fiat-Shamir, as no public message is available. Instead, we let $\mathcal{P}$ distribute the shares of a random sharing $[r]$, and then broadcast a public message $u = c_1 + r$ to all verifiers, who can compute $[c_1] = u - [r]$. The verifiers can locally compute $[c_2] = [z] - [c_1]$.

4. We adopt the polynomial approach to compress two inner-product tuples $([\boldsymbol{a}_1], [\boldsymbol{b}_1], [c_1])$ and $([\boldsymbol{a}_2], [\boldsymbol{b}_2], [c_2])$ into a single tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$, which has been used in prior work [BFO12, NV18, GS20, GSZ20]. Differently, we will use the Fiat-Shamir transform to realize the non-interactive compression. Specifically, the parties compute the sharings of polynomials $[\boldsymbol{f}(\cdot)]$, $[\boldsymbol{g}(\cdot)]$ and $[h(\cdot)]$, such that $\boldsymbol{f}(i) = \boldsymbol{a}_i$, $\boldsymbol{g}(i) = \boldsymbol{b}_i$ and $h(i) = c_i$ for $i \in [1, 2]$. $\mathcal{P}$ needs to convince the verifiers that $h(X) = \boldsymbol{f}(X) \odot \boldsymbol{g}(X)$, which can be realized by proving $h(\alpha) = \boldsymbol{f}(\alpha) \odot \boldsymbol{g}(\alpha)$ for a random challenge α. $\mathcal{P}$ and all verifiers can generate α by computing $\mathsf{H}'(\gamma, msg)$ where γ is the challenge used in the previous iteration and msg is the public message sent in the current iteration. Now, the parties can define $[\boldsymbol{x}] = [\boldsymbol{f}(\alpha)]$, $[\boldsymbol{y}] = [\boldsymbol{g}(\alpha)]$ and $[z] = [h(\alpha)]$, and execute the next iteration.

5. Let $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ be the inner-product tuple with dimension 2 after the dimension reduction is completed. We can adopt the randomization technique [BBC$^+$19, GSZ20] to check the correctness of this tuple. In the same way, we can split it into two multiplication triples $([x_1], [y_1], [z_1])$ and $([x_2], [y_2], [z_2])$. The prover $\mathcal{P}$ can distribute the shares of a random multiplication triple $([x_0], [y_0], [z_0])$ with $z_0 = x_0 \cdot y_0$ to all verifiers in the way compatible with Fiat-Shamir. Then, $\mathcal{P}$ and the verifiers can compress $\{([x_i], [y_i], [z_i])\}_{i\in[0,2]}$ into $([x], [y], [z])$. All verifiers can run the Open procedure to obtain (x, y, z) and check that $z = x \cdot y$.

Streaming strong NIMVZK with the same round complexity. We can use the strong NIMVZK protocol to prove a very large circuit in a streamable way, such that between the prover and verifiers are non-interactive for proving a batch of N multiplication gates each time, and the verifiers *still* communicate only one round for proving the whole circuit. For a batch of N multiplication gates, the parties can transform the sharings into an inner-product tuple with dimension N, and then compress it into an inner-product tuple $([\boldsymbol{x}_1], [\boldsymbol{y}_1], [z_1])$ with dimension $M = N/2^c$ for some integer $c \geq 1$. For another batch of multiplication gates, the parties can generate another inner-product tuple $([\boldsymbol{x}_2], [\boldsymbol{y}_2], [z_2])$ with dimension $M = N/2^c$ in the same way. Then, the prover and verifiers can compress $([\boldsymbol{x}_1], [\boldsymbol{y}_1], [z_1])$ and $([\boldsymbol{x}_2], [\boldsymbol{y}_2], [z_2])$ into an inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with the same dimension M, where the challenge α for this compression is computed with random oracle H' and two challenges to obtain the tuples $([\boldsymbol{x}_1], [\boldsymbol{y}_1], [z_1])$ and $([\boldsymbol{x}_2], [\boldsymbol{y}_2], [z_2])$. After the whole circuit has been evaluated, the verifiers can check correctness of the final inner-product tuple (with dimension M) stored in memory by communicating only one round. As a result, all parties only need memory linear to what is needed to evaluate the statement in the clear.

3.4 More Efficient Strong NIMVZK from Packed Secret Sharing

The above discussion shows a strong NIMVZK protocol where a prover sends one message to each verifier and the verifiers communicate only one round. It is secure against the adversary corrupting up to a minority of verifiers (i.e., $t < n/2$) and the prover. The protocol is also streamable meaning that the prover can send the proof on-the-fly. However, the downside is the communication of $1/2 + o(1)$ field elements per multiplication gate per verifier and a majority of the proof is used to transmit the shares of wire values in the circuit. We now discuss the strong NIMVZK protocol that reduces the communication cost to $O(1/n)$ field elements per multiplication gate when the threshold of corrupted verifiers $t < n(1/2 - \epsilon)$ for any $0 < \epsilon < 1/2$. This protocol adopts packed secret sharing (PSS) [FY92] as the underlying LSSS, where each sharing packs $k = O(n)$ secrets.

The verification idea described as above for strong NIMVZK on inner-product tuples can be generalized easily to the setting of packed secret sharing and used to check the correctness of a packed inner-product tuple. However, using packed secret sharing efficiently for a single generic circuit is a huge challenge, because the layout of the circuit could be complicated for packing k gates. In fact, because of this, prior MPC work [GSY21, BGJK21] using PSS focus on SIMD operations (i.e., repeated circuits). For a single generic circuit, the state-of-the-art PSS-based MPC protocol [GPS21] requires to evaluate the circuit layer-by-layer that needs the rounds linear to the circuit depth and split each output wire into different output wires that each can be used only once. Fortunately, we observe that even a single generic circuit can be packed optimally in the context of zero-knowledge, and can remove the constraints in MPC. Particularly, the prover can prove a circuit in a streamable way without the constraint of proving the circuit layer-by-layer, as the prover knows all the wire values.

The greatest challenge for adopting PSS is how to store the wire values in packed sharings, which allows to evaluate the circuit without changing the circuit output and enable the packed sharings to be checked using an efficient verification technique. Without packing, this is easy because the parties can move around any individual wire. However, with PSS, it is not possible. In our NIMVZK protocol, if the out degree of a gate is greater than 1, we allow the output wire to appear multiple times (instead of splitting the output wire into multiple output wires), which enables to obtain better communication. In this case, we need to use the consistency check to ensure that the same wire is assigned with the same value. Specifically, for each input packed sharing $[\boldsymbol{y}]$, if the j-th secret y_j comes from the i-th secret x_i stored in an output packed sharing $[\boldsymbol{x}]$, then we need to check $x_i = y_j$. This corresponds to the wire that carries the value $x_i = y_j$. Following the work [GPS21], we refer to $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$ as a wire tuple. For the consistency check of wire tuples, we reduce the *total* communication complexity from $O(n^5k^2)$ in MPC [GPS21] to $O(n^2k^2)$ for our strong NIMVZK protocol. For each $i, j \in [1, k]$, let $([\boldsymbol{x}_1], [\boldsymbol{y}_1], i, j), \ldots, ([\boldsymbol{x}_m], [\boldsymbol{y}_m], i, j)$ be the wire tuples with the same indices i, j. We use the random-linear-combination approach to check the consistency. Specifically, the prover $\mathcal{P}$ samples two random vectors $\boldsymbol{x}_0, \boldsymbol{y}_0$ such that $x_{0,i} = y_{0,j}$, and then distributes the shares of $[\boldsymbol{x}_0]$ and $[\boldsymbol{y}_0]$ to all verifiers. To support Fiat-Shamir, we need $\mathcal{P}$ to generate these shares in two steps: 1) distributing the shares of two random sharings $[\boldsymbol{r}]$ and $[\boldsymbol{s}]$; and 2) broadcasts the differences $\boldsymbol{u} = \boldsymbol{x}_0 + \boldsymbol{r}$ and $\boldsymbol{v} = \boldsymbol{y}_0 + \boldsymbol{s}$ to all verifiers. Then the verifiers can compute $[\boldsymbol{x}_0] := \boldsymbol{u} - [\boldsymbol{r}]$ and $[\boldsymbol{y}_0] = \boldsymbol{v} - [\boldsymbol{s}]$. $\mathcal{P}$ and all verifiers can generate a random challenge $\alpha = \mathsf{H}'(\chi, \boldsymbol{u}, \boldsymbol{v}, i, j)$, where χ is a public value related to the secrets $\{(\boldsymbol{x}_h, \boldsymbol{y}_h)\}_{h \in [1,m]}$. Then, the verifiers can now check correctness of the following wire tuple:

$$[\boldsymbol{x}] := \sum_{h=1}^{m} \alpha^h \cdot [\boldsymbol{x}_h] + [\boldsymbol{x}_0], \; [\boldsymbol{y}] := \sum_{h=1}^{m} \alpha^h \cdot [\boldsymbol{y}_h] + [\boldsymbol{y}_0].$$

This check can be done by letting the verifiers open $([\boldsymbol{x}], [\boldsymbol{y}])$ and check $x_i = y_j$. When streaming the strong NIMVZK protocol, the verification of wire tuples do *not* increase the rounds between verifiers (see Section 6.1 for details).

4 Information-Theoretic NIMVZK Proof in the Honest-Majority Setting

We present a non-interactive multi-verifier zero-knowledge (NIMVZK) protocol with information-theoretic security in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{verifyprod}})$-hybrid model, assuming an honest majority of verifiers, where $\mathcal{F}_{\mathsf{coin}}$ is a coin-tossing functionality shown in Appendix A. Functionality $\mathcal{F}_{\mathsf{verifyprod}}$ allows to verify the correctness of

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

This functionality runs with a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$. Let $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ be an inner-product tuple defined over a field $\mathbb{K}$, where the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is N. This functionality operates as follows:

1. Upon receiving the shares of $[\boldsymbol{x}], [\boldsymbol{y}], [z]$ from honest verifiers, execute the following:
   - Reconstruct the secret vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^N$ and secret $z \in \mathbb{F}$ from these shares.
   - Compute the shares of $[\boldsymbol{x}], [\boldsymbol{y}], [z]$ held by corrupted verifiers, and send them to the adversary.
   - If $\mathcal{P}$ is corrupted, send $(\boldsymbol{x}, \boldsymbol{y}, z)$ to the adversary.
2. If $z \neq \boldsymbol{x} \odot \boldsymbol{y}$, then set $b := \mathsf{abort}$, otherwise set $b := \mathsf{accept}$. Send b to the adversary. For each $i \in \mathcal{H}$, wait for an input from the adversary, and then do the following:
   - If it is $\mathsf{continue}_i$, send b to $\mathcal{V}_i$.
   - If it is abort_i, send abort to $\mathcal{V}_i$.

Figure 2: **Zero-knowledge verification functionality for an inner-product tuple.**

an inner-product tuple secretly shared among verifiers. It is possible to instantiate $\mathcal{F}_{\mathsf{verifyprod}}$ using prior work on fully linear PCP (or IOP) [BBC+19], but we can improve its communication (or rounds) by adapting the technique by Goyal et al. [GS20, GSZ20] in the MPC setting to the MVZK setting.

4.1 From General Adversaries to Maximal Adversaries for MVZK

Before we describe the NIMVZK protocol, we prove an important lemma that can be used to simplify the proofs of the MVZK protocols in this paper and the future works. Informally, this lemma states that if an MVZK protocol is secure against *exactly* t malicious verifiers, then the protocol is also secure against *at most* t malicious verifiers. The proof of this lemma is based on that of a similar lemma for honest-majority MPC by Genkin et al. [GIP+14]. This lemma allows us to only consider the maximum adversaries who corrupt exactly t verifiers, and thus simplifies the security proofs of MVZK protocols. One caveat is that the proof of this lemma needs to specially deal with the case that the honest verifiers will receive output as well as the possible random-oracle queries (e.g., the Fiat-Shamir transform [FS87] is used).

Lemma 1. *Let Π be an MVZK protocol proving the satisfiability of a circuit C for $n \geq 2t+1$ verifiers. Then, if protocol Π securely realizes $\mathcal{F}_{\mathsf{mvzk}}$ in the presence of any malicious adversary corrupting exactly t verifiers, then Π securely realizes $\mathcal{F}_{\mathsf{mvzk}}$ against any malicious adversary corrupting at most t verifiers.*

The proof of the above lemma is given in Appendix B. The above lemma can be applied to not only our information-theoretic NIMVZK protocol but also the strong NIMVZK proofs in the computational setting that will be described in Section 5 and Section 6.

4.2 Our Information-Theoretic NIMVZK Protocol

In Figure 3, we describe the detailed NIMVZK protocol with information theoretic security in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{verifyprod}})$-hybrid model. For each circuit-input gate or multiplication gate, the prover directly shares the output value to all verifiers. The verifiers can locally compute the shares on the output wires of addition gates. Then, all verifiers check the correctness of all multiplication gates by transforming multiplication triples into an inner-product tuple and then calling functionality $\mathcal{F}_{\mathsf{verifyprod}}$. In parallel, the verifiers also check correctness of the single circuit-output gate via running the Open procedure.

Protocol $\Pi^{\mathsf{it}}_{\mathsf{nimvzk}}$

Inputs: A prover $\mathcal{P}$ holds a witness $\boldsymbol{w} \in \mathbb{F}^m$. $\mathcal{P}$ and all verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold an arithmetic circuit C over a field $\mathbb{F}$ with $|\mathbb{F}| > n$. Let N denote the number of multiplication gates in the circuit. $\mathcal{P}$ will convince the verifiers that $C(\boldsymbol{w}) = 0$.

Circuit evaluation: In a predetermined topological order, $\mathcal{P}$ and all verifiers evaluate the circuit as follows:

- For each circuit-input wire with input value $w \in \mathbb{F}$, $\mathcal{P}$ (acting as the dealer) runs $[w] \leftarrow \mathsf{Share}(w)$ to distribute the shares to all verifiers.
- For each addition gate with input sharings $[x]$ and $[y]$, all verifiers locally compute $[z] := [x] + [y]$, and $\mathcal{P}$ computes $z := x + y \in \mathbb{F}$.
- For each multiplication gate with input values $x, y \in \mathbb{F}$, $\mathcal{P}$ computes $z := x \cdot y \in \mathbb{F}$, and then executes $[z] \leftarrow \mathsf{Share}(z)$ which distributes the shares to all verifiers.

Verification of multiplication gates: Let $([x_i], [y_i], [z_i])$ be the sharings on the i-th multiplication gate for $i \in [1, N]$. All verifiers and $\mathcal{P}$ execute as follows:

1. The verifiers call the coin-tossing functionality $\mathcal{F}_{\mathsf{coin}}$ to generate a random element $\chi \in \mathbb{K}$, and then set the following inner-product tuple:
$$[\boldsymbol{x}] := \left([x_1], \chi \cdot [x_2], \ldots, \chi^{N-1} \cdot [x_N]\right), \ [\boldsymbol{y}] := \left([y_1], [y_2], \ldots, [y_N]\right), \ [z] := \textstyle\sum_{i \in [1,N]} \chi^{i-1} \cdot [z_i].$$
2. The verifiers and $\mathcal{P}$ call functionality $\mathcal{F}_{\mathsf{verifyprod}}$ on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to check that $z = \boldsymbol{x} \odot \boldsymbol{y}$. If the verifiers receive abort from $\mathcal{F}_{\mathsf{verifyprod}}$, then they abort.

Verification of circuit output: Let $[\eta]$ be the input sharing associated with the single circuit-output gate. All verifiers execute $\eta \leftarrow \mathsf{Open}([\eta])$, and abort if outputting abort in the Open procedure. If $\eta = 0$, then the verifiers output true, otherwise they output false.

Figure 3: **Information-theoretic NIMVZK protocol in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{verifyprod}})$-hybrid model.**

In Figure 2, we give the precise definition of functionality $\mathcal{F}_{\mathsf{verifyprod}}$. In particular, if the prover is honest, the adversary can only obtain the shares of corrupted verifiers from this functionality, which does not reveal any information on the secrets. In other words, this functionality naturally captures zero-knowledge. If the prover is corrupted, this functionality reveals all secrets to the adversary, as the secrets have been known anyway by the adversary. We can view deciding the correctness of an inner-product tuple as a statement which is shared among n verifiers. Functionality $\mathcal{F}_{\mathsf{verifyprod}}$ guarantees that the malicious prover cannot make any honest verifier accept a false statement, and thus captures soundness. In Appendix C, we present an efficient protocol to securely realize $\mathcal{F}_{\mathsf{verifyprod}}$, where the communication and round complexities are $O((n+\tau)\log_\tau |C|)$ field elements per verifier and $\log_\tau |C| + 3$ rounds between verifiers respectively, where $\tau \geq 2$ is a parameter.

Theorem 1. *Protocol $\Pi^{\mathsf{it}}_{\mathsf{nimvzk}}$ shown in Figure 3 securely realizes functionality $\mathcal{F}_{\mathsf{mvzk}}$ with information-theoretic security and soundness error $\frac{N-1}{|\mathbb{K}|}$ in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{verifyprod}})$-hybrid model in the presence of a malicious adversary corrupting up to a prover and t verifiers.*

The proof of this theorem can be found in Appendix C.

Protocol $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$

Inputs: A prover $\mathcal{P}$ holds a witness $\boldsymbol{w} \in \mathbb{F}^m$. $\mathcal{P}$ and the verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold an arithmetic circuit C over a field $\mathbb{F}$ with $|\mathbb{F}| > n$ such that $C(\boldsymbol{w}) = 0$. Let N denote the number of multiplication gates in the circuit. Let $\mathsf{H}_1 : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{H}_2 : \{0,1\}^* \to \mathbb{K}$ be two random oracles.

Circuit evaluation: $\mathcal{P}$ and all verifiers evaluate the circuit in the same way as described in Figure 3.

Verification of multiplication gates: For all m circuit-input wires, the verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold the shares of $[w_1], \ldots, [w_m]$, and $\mathcal{P}$ has the witness $\boldsymbol{w} = (w_1, \ldots, w_m)$. For all N multiplication gates, $\mathcal{P}$ and the verifiers respectively hold the secrets and the shares of multiplication triples $([x_1], [y_1], [z_1]), \ldots, ([x_N], [y_N], [z_N])$. For $j \in [1, m]$, let $w_j^1, \ldots, w_j^n$ be the shares of $[w_j]$ held by all verifiers, which are also known by $\mathcal{P}$. For $j \in [1, N]$, let $z_j^1, \ldots, z_j^n$ be the shares of $[z_j]$, which are also obtained by $\mathcal{P}$. All verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ and $\mathcal{P}$ execute as follows:

1. For each $i \in [1, n]$, $\mathcal{P}$ samples $r_i \leftarrow \{0,1\}^\lambda$ and computes
$$com_i := \mathsf{H}_1(w_1^i, \ldots, w_m^i, z_1^i, \ldots, z_N^i, r_i).$$
Then, $\mathcal{P}$ broadcasts $(com_1, \ldots, com_n)$ to all verifiers, and also sends r_i to every verifier $\mathcal{V}_i$ over a private channel.
2. Every verifier $\mathcal{V}_i$ checks that $com_i = \mathsf{H}_1(w_1^i, \ldots, w_m^i, z_1^i, \ldots, z_N^i, r_i)$, and aborts if the check fails.
3. $\mathcal{P}$ and all verifiers compute $\chi := \mathsf{H}_2(com_1, \ldots, com_n) \in \mathbb{K}$.
4. $\mathcal{P}$ and all verifiers respectively compute the secrets and the shares of the following inner-product tuple:
$$[\boldsymbol{x}] := ([x_1], \chi \cdot [x_2], \ldots, \chi^{N-1} \cdot [x_N]),\ [\boldsymbol{y}] := ([y_1], [y_2], \ldots, [y_N])\,,\ [z] := \textstyle\sum_{i \in [1,N]} \chi^{i-1} \cdot [z_i].$$
5. The verifiers and $\mathcal{P}$ call functionality $\mathcal{F}_{\mathsf{verifyprod}}$ on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to check that $z = \boldsymbol{x} \odot \boldsymbol{y}$. If the verifiers receive abort from $\mathcal{F}_{\mathsf{verifyprod}}$, then they abort.

Verification of circuit output: All verifiers check the correctness of a single circuit-output gate in the same way as shown in Figure 3.

Figure 4: **Strong non-interactive MVZK protocol in the $\mathcal{F}_{\mathsf{verifyprod}}$-hybrid model and random oracle model.**

5 Strong NIMVZK Proof in the Honest-Majority Setting

In this section, we present a strong NIMVZK proof based on the Fiat-Shamir transform, where a minority of verifiers are allowed to be corrupted and collude with the prover. Recall that non-interactive means that the prover only sends one message to every verifier, and a verifier only sends one message to every other verifier. Our strong NIMVZK protocol adopts a non-interactive commitment based on random oracle to non-interactively transform the verification of multiplication triples into the verification of an inner-product tuple. This protocol still works in the $\mathcal{F}_{\mathsf{verifyprod}}$-hybrid model, where functionality $\mathcal{F}_{\mathsf{verifyprod}}$ can now be non-interactively realized using the Fiat-Shamir transform.

In Figure 4, we describe the strong NIMVZK protocol $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$ in the $\mathcal{F}_{\mathsf{verifyprod}}$-hybrid model, where the shares are computed over a field $\mathbb{F}$ and the verification of multiplication gates is performed over an extension field $\mathbb{K}$ with $|\mathbb{K}| \geq 2^\lambda$. The strong NIMVZK protocol is the same as the protocol shown in Figure 3, except for the verification of multiplication gates. In the strong NIMVZK protocol, the verification of multiplication gates is executed non-interactively using a non-interactive commitment based on a random oracle H_1, where a commitment com on a message x is defined as $\mathsf{H}_1(x, r)$ for a randomness $r \in \{0,1\}^\lambda$. However, we do *not* require that the commitment is binding. Instead, we only need the commitment to be hard to find a pair

(x', r') such that $\mathsf{H}_1(x', r') = \mathsf{H}_1(x, r)$ and $x' \neq x$, after $\mathsf{H}_1(x, r)$ has been defined. This has been explained in Section 3.3 (see the proof of Theorem 2 for details). The random challenge $\chi \in \mathbb{K}$ is now generated using another random oracle H_2 and the public commitments, instead of calling $\mathcal{F}_{\mathsf{coin}}$. In this case, the prover can compute the secrets $(\boldsymbol{x}, \boldsymbol{y}, z)$ underlying the inner-product tuple using the public coefficient χ and the secret wire values. At first glance, the secrets $(\boldsymbol{x}, \boldsymbol{y}, z)$ seem to be useless for the protocol execution of $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$. Nevertheless, the prover can use $(\boldsymbol{x}, \boldsymbol{y}, z)$ to compute all the secrets involved in the protocol that securely realizes functionality $\mathcal{F}_{\mathsf{verifyprod}}$. In this case, we can securely compute $\mathcal{F}_{\mathsf{verifyprod}}$ in a strongly non-interactive way by making the prover distribute the shares of all secrets non-interactively and all verifiers interact only one round for Open.

Theorem 2. *Let H_1 and H_2 be two random oracles. Protocol $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$ shown in Figure 4 securely realizes functionality $\mathcal{F}_{\mathsf{mvzk}}$ with soundness error at most $\frac{Q_1 n + (Q_2+1)N}{2^\lambda}$ in the $\mathcal{F}_{\mathsf{verifyprod}}$-hybrid model in the presence of a malicious adversary corrupting up to a prover and t verifiers, where Q_1 and Q_2 are the number of queries to random oracles H_1 and H_2 respectively.*

The proof of the above theorem is given in Appendix D.

Optimizations. We can further optimize the strong NIMVZK protocol $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$ shown in Figure 4 as follows:

1. For Shamir secret sharing defined in Appendix A.2, in the computational setting, the prover $\mathcal{P}$ can send a random $\mathsf{seed}_i \in \{0,1\}^\lambda$ to a verifier $\mathcal{V}_i$ for each $i \in [1, t]$, who computes all its shares with seed_i and a pseudo-random generator (PRG). This reduces the communication by a half. Furthermore, for each $i \in [1, t]$, $\mathcal{P}$ can send $com_i = \mathsf{H}_1(\mathsf{seed}_i, r_i)$ to $\mathcal{V}_i$, who checks the correctness of com_i using seed_i and r_i. This optimization will reduce the computational cost of generating and verifying t commitments.

   Using the above optimization, the communication among verifiers is *asymmetry*. In particular, among all verifiers, t verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_t$ only has a *sublinear* communication complexity. That is, each verifier only receives $O(n + \log |C|)$ field elements from the prover, and only needs to send $O(n)$ field elements to other verifiers. It makes our strong NIMVZK protocol particularly suitable for the applications where $\mathcal{V}_1, \ldots, \mathcal{V}_t$ are lower-resource mobile devices and the other verifiers are powerful servers.

2. The prover and all verifiers can use $\mathsf{H}_2(com_1, \ldots, com_n)$ to compute random challenges $\chi_1, \ldots, \chi_N \in \mathbb{K}$, and then use these challenges to transform the verification of N multiplication triples into that of an inner-product tuple. In particular, $[\boldsymbol{x}] := (\chi_1 \cdot [x_1], \ldots, \chi_N \cdot [x_N])$ and $[z] := \sum_{i \in [1,N]} \chi_i \cdot [z_i]$ where $[\boldsymbol{y}]$ is defined in the same way as shown in Figure 4. In this way, the soundness error can be reduced from $\frac{Q_1 n + (Q_2+1)N}{2^\lambda}$ to $\frac{Q_1 n + Q_2 + 1}{2^\lambda}$, as independent random coefficients $\chi_1, \chi_2, \ldots, \chi_N$ instead of $1, \chi, \ldots, \chi^{N-1}$ are used.

Strong NIMVZK proof for inner-product tuples. Boneh et al. [BBC+19] introduced a powerful tool, called distributed zero-knowledge (DZK) proof (a.k.a., ZK proof on a distributed or secret-shared statement), to prove the inner-product statements (and other useful statements). We can use their DZK proof with logarithmic communication to securely realize functionality $\mathcal{F}_{\mathsf{verifyprod}}$ shown in Figure 2. When applying the Fiat-Shamir transform [BGIN20] into their DZK proof, the prover non-interactively sends a proof to all verifiers, and the verifiers execute one-round communication to verify correctness of an inner-product tuple. Note that the proof on the inner-product statement can be sent in parallel with our proof on circuit satisfiability shown in Figure 4. Therefore, using the DZK proof to instantiate $\mathcal{F}_{\mathsf{verifyprod}}$, our MVZK protocol is strongly non-interactive. While Boneh et al. [BBC+19] originally instantiated the DZK proof with replicated secret sharing, Boyle et al. [BGIN20] shown that their DZK proof also works for verifiable Shamir secret sharing meaning that a consistency check is needed to guarantee either all verifiers hold a consistent sharing of the secret or honest verifiers abort.

We can simplify the technique by Boneh et al. [BBC+19] by avoiding the use of a verifiable secret sharing scheme, and slightly optimize the communication from $4.5 \log |C| + 5n$ field elements to $3 \log |C| + 3n$ field elements. We can also improve the hash computation cost for Fiat-Shamir. In Section 3.3, we gave an overview of our approach for proving correctness of an inner-product tuple, and thus omit it here. The detailed protocol to strongly non-interactively realize $\mathcal{F}_{\mathsf{verifyprod}}$ can be directly obtained by simplifying the PSS-based protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ shown in Figure 8 of Section 6.2 via setting the number of packed secrets $k = 1$.

6 Strong NIMVZK with Lower Communication from PSS

Based on packed secret sharing (PSS), we present a strong NIMVZK proof with communication complexity $O(|C|/n)$ per verifier, when the threshold of corrupted verifiers $t < n(1/2 - \epsilon)$ for any $0 < \epsilon < 1/2$. Our strong NIMVZK protocol is highly efficient for proving satisfiability of a *single generic* circuit. In the ZK setting, we use PSS optimally. In particular, we eliminate the constraints in the state-of-the-art PSS-based MPC protocol [GPS21] including: 1) evaluating a circuit layer-by-layer, 2) interactively permuting the secrets in a single packed sharing, 3) interactively collecting the secrets from different packed sharings and 4) splitting an output wire into multiple output wires, where all these constraints will make the rounds and communication cost significantly larger than our protocol.

Firstly, we discuss how to transform a general circuit C into another circuit C' with the same output and $|C'| = |C| + O(k)$, such that 1) the number of circuit-input wires, addition gates and multiplication gates is the multiple of k; 2) there are at least k circuit-output wires; 3) the gates with the same type are divided into groups of k. This is done by adding "dummy" wires and gates, and is described in Appendix E. Then, we present the detailed strong NIMVZK protocol in the $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$-hybrid model, where $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ verifies the correctness of a packed inner-product tuple. Next, we present a strong non-interactive MVZK protocol to securely realize functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$.

6.1 Strong NIMVZK based on Packed Secret Sharing

Before showing the detailed strong NIMVZK protocol, we give the definition of functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$.

Functionality for verifying packed inner-product tuples. Let $\mathcal{H}_H \subset \mathcal{H}$ be a fixed $(d+1)$-sized subset of honest verifiers and $\mathcal{H}_C = \mathcal{H} \backslash \mathcal{H}_H$, where recall that $\mathcal{H}$ is the set of all $d + k$ honest verifiers and d is the degree of polynomials for PSS. For a degree-d packed sharing $[\boldsymbol{x}]$, we use $[\boldsymbol{x}]_{\mathcal{H}}$ to denote the whole sharing that is reconstructed from the shares of honest verifiers in $\mathcal{H}_H$. Given the shares of honest verifiers in $\mathcal{H}$ as input, we can reconstruct the *whole* sharing $[\boldsymbol{x}]$ as follows:

1. Use the $d+1$ shares of honest verifiers in $\mathcal{H}_H$ to reconstruct the whole sharing $[\boldsymbol{x}]_{\mathcal{H}}$. Define the shares of corrupted verifiers on $[\boldsymbol{x}]$ as that on $[\boldsymbol{x}]_{\mathcal{H}}$. Following prior MPC work [GPS21], we always assume that the corrupted verifiers in $\mathcal{C}$ hold the correct shares that they should hold, while they may use incorrect shares during the protocol execution, where $\mathcal{C}$ is the set of corrupted verifiers.

2. Define the secrets of $[\boldsymbol{x}]$ to be that of $[\boldsymbol{x}]_{\mathcal{H}}$.

3. Define the shares of $[\boldsymbol{x}]$ held by honest verifiers in $\mathcal{H}$ as the shares input by the verifiers directly.

The zero-knowledge verification functionality for packed inner-product tuples is shown in Figure 5. This functionality takes as input a packed inner-product tuple and then checks correctness of the tuple, where each sharing packs k secrets. This functionality sends the shares of corrupted verifiers for each packed sharing to the adversary, where the shares are computed by the above approach based on the shares of honest verifiers

Functionality $\mathcal{F}_{\mathsf{verifyprod}}^{\mathsf{pss}}$

The packed inner-product tuple over a field $\mathbb{K}$ is denoted by $([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_\ell])$, $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_\ell])$ and $[\boldsymbol{z}]$. This functionality runs with a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$, and operates as follows:

1. Upon receiving the shares of $\{[\boldsymbol{x}_i], [\boldsymbol{y}_i]\}_{i \in [1,\ell]}$ and $[\boldsymbol{z}]$ from all honest verifiers, execute the following:
   - Reconstruct the secrets $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell)$, $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_\ell)$ and $\boldsymbol{z}$ from the shares of honest verifiers in $\mathcal{H}_H$.
   - Compute the shares of corrupted verifiers on $([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_\ell])$, $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_\ell])$ and $[\boldsymbol{z}]$, and then send these shares to the adversary.
   - If $\mathcal{P}$ is corrupted, send the shares of $\left(\{[\boldsymbol{x}_i], [\boldsymbol{y}_i]\}_{i \in [1,\ell]}, [\boldsymbol{z}]\right)$ held by honest verifiers in $\mathcal{H}$ and the secrets $\left(\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i \in [1,\ell]}, \boldsymbol{z}\right)$ to the adversary.
2. If $\boldsymbol{z} \neq \sum_{h \in [1,\ell]} \boldsymbol{x}_h * \boldsymbol{y}_h$ where $*$ denotes the component-wise product, then set $b := \mathsf{abort}$, otherwise set $b := \mathsf{accept}$. Send b to the adversary. For $i \in \mathcal{H}$, wait for an input from the adversary, and do the following:
   - If it is $\mathsf{continue}_i$, send b to $\mathcal{V}_i$.
   - If it is abort_i, send abort to $\mathcal{V}_i$.

Figure 5: **ZK verification functionality for packed inner-product tuples.**

in $\mathcal{H}_H$. If the prover is corrupted, this functionality also sends the shares of all honest verifiers and the secrets in all packed sharings to the adversary, as these shares and secrets have been known by the adversary.

PSS-based strong NIMVZK protocol from the Fiat-Shamir transform. Our PSS-based strong non-interactive MVZK protocol in the $\mathcal{F}_{\mathsf{verifyprod}}^{\mathsf{pss}}$-hybrid model is described in Figures 6 and 7, where the circuit is defined over a field $\mathbb{F}$ and the verification is performed over an extension field $\mathbb{K}$ with $|\mathbb{K}| \geq 2^\lambda$.

The prover and all verifiers first transform the circuit C into an *equivalent* circuit C', which satisfies the requirements of packed secret sharings. For an input vector $\boldsymbol{w} \in \mathbb{F}^k$, if the j-th secret of $\boldsymbol{w}$ for $j \in [1, k]$ corresponds to a dummy circuit-input wire, the secret is set as 0. If $\boldsymbol{w}$ corresponds to k dummy circuit-input wires, then $\boldsymbol{w} = 0^k$ and $[\boldsymbol{w}]$ can be *locally* generated by all verifiers without any communication (as shown in Appendix A.2).

Using the non-interactive commitment based on a random oracle, we adopt a similar approach as described in the previous section to transform the check of multiplication tuples into the check of a packed inner-product tuple. Then, by calling functionality $\mathcal{F}_{\mathsf{verifyprod}}^{\mathsf{pss}}$, the verifiers can check correctness of the packed inner-product tuple. Note that the prover can compute the secrets of the packed inner-product tuple, which will be useful for designing a strongly non-interactive protocol to securely realize $\mathcal{F}_{\mathsf{verifyprod}}^{\mathsf{pss}}$ as shown in Section 6.2.

During the protocol execution, we need to check the consistency of some secrets stored in two different packed sharings. We perform the consistency check using the random-linear-combination approach based on the Fiat-Shamir transform, which is inspired by the recent checking approach by Goyal et al. [GPS21] for information-theoretic MPC. In particular, the prover will generate a packed input sharing $[\boldsymbol{y}]$ on k multiplication gates, addition gates or circuit-output gates, such that the j-th secret y_j of $[\boldsymbol{y}]$ comes from the i-th secret x_i of a packed output sharing $[\boldsymbol{x}]$ of k circuit-input gates, multiplication gates or addition gates. We need to check that $x_i = y_j$ to guarantee the consistency of y_j. This corresponds to the wire which carries the value $x_i = y_j$ in the circuit. We refer to a tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$ as a *wire tuple* following prior work [GPS21]. We perform the consistency check of wire tuples in the *total* communication complexity $O(nk^2)$ elements between the prover and verifiers and $O(n^2k^2)$ elements among verifiers.

Theorem 3. *Let* H_1 *and* H_2 *be two random oracles. Protocol* $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$ *shown in Figures 6 and 7 securely*

Protocol $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$

Inputs: A prover $\mathcal{P}$ holds a witness $\bar{\boldsymbol{w}}$. $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold a circuit C over a field $\mathbb{F}$. Let k denote the number of secrets that are packed in a single sharing. Let m be the number of packed sharings on circuit-input wires, where each packs k circuit-input gates. Let $M = \lceil |C|/k \rceil$ denote the number of multiplication tuples where each packs k multiplication gates. Let $N = O(|C|)$ represent the number of wire tuples in the form of $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$ with $x_i = y_j$. Let $\mathsf{H}_1 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ and $\mathsf{H}_2 : \{0,1\}^* \rightarrow \mathbb{K}$ be two random oracles.

Preprocess circuit: All parties run the $\mathsf{PrepCircuit}$ procedure shown in Figure 16 of Appendix E to preprocess the circuit C, and obtain an equivalent circuit C' such that $C'(\bar{\boldsymbol{w}}) = C(\bar{\boldsymbol{w}})$ for any input $\bar{\boldsymbol{w}}$.

Circuit evaluation: In a predetermined topological order, $\mathcal{P}$ and all verifiers evaluate the circuit C' as follows:

- Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m \in \mathbb{F}^k$ be the secret vectors where each associates with k circuit-input wires. For each group of k circuit-input wires with an input vector $\boldsymbol{w} \in \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\}$, $\mathcal{P}$ runs $[\boldsymbol{w}] \leftarrow \mathsf{Share}(\boldsymbol{w})$ to distribute the shares to all verifiers.
- For each group of k addition gates with input sharings $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$, all verifiers *locally* compute $[\boldsymbol{z}] := [\boldsymbol{x}] + [\boldsymbol{y}]$, and $\mathcal{P}$ computes $\boldsymbol{z} := \boldsymbol{x} + \boldsymbol{y} \in \mathbb{F}^k$.
- For each group of k multiplication gates with input sharings $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$, $\mathcal{P}$ computes $\boldsymbol{z} := \boldsymbol{x} * \boldsymbol{y} \in \mathbb{F}^k$ where $*$ denotes the component-wise product, and executes $[\boldsymbol{z}] \leftarrow \mathsf{Share}(\boldsymbol{z})$ which distributes the shares to all verifiers.
- For each group of k input wires of multiplication, addition, or circuit-output gates, such that the corresponding input vector $\boldsymbol{y} \in \mathbb{F}^k$ has not been stored in any single packed input sharing, $\mathcal{P}$ runs $[\boldsymbol{y}] \leftarrow \mathsf{Share}(\boldsymbol{y})$, which distributes the shares to all verifiers.

Locally prepare packed sharings. All verifiers locally do the following:

- For each input sharing $[\boldsymbol{y}]$ on a group of k multiplication, addition, or circuit-output gates, for each position $j \in [1,k]$, suppose that y_j comes from the i-th secret of $[\boldsymbol{x}]$ that is an output sharing on a group of k circuit-input, multiplication, or addition gates. If $[\boldsymbol{y}] \neq [\boldsymbol{x}]$, then set a wire tuple as $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$.
- Denote these wire tuples by $([\boldsymbol{x}_1], [\boldsymbol{y}_1], i_1, j_1), \ldots, ([\boldsymbol{x}_N], [\boldsymbol{y}_N], i_N, j_N)$.
- Remove the repetitive packed sharings in $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_N]$, and then denote the resulting sharings as $[\boldsymbol{y}'_1], \ldots, [\boldsymbol{y}'_\ell]$, where ℓ denotes the number of different packed sharings in $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_N]$.

All verifiers hold the shares of the following packed sharings:

- The shares of $[\boldsymbol{w}_1], \ldots, [\boldsymbol{w}_m]$ on all circuit-input wires, which are denoted by $(\hat{w}_i^1, \ldots, \hat{w}_i^n)$ for $i \in [1,m]$.
- Let $([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])$ be the i-th multiplication tuple packing the secrets of k multiplication gates for $i \in [1,M]$. The shares of $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i \in [1,M]}$, where $\hat{z}_i^1, \ldots, \hat{z}_i^n$ denote the shares of $[\boldsymbol{z}_i]$ for $i \in [1,M]$.
- The shares of $\{[\boldsymbol{y}'_i]\}_{i \in [1,\ell]}$, which are denoted by $\hat{y}_i^1, \ldots, \hat{y}_i^n$ for $i \in [1,\ell]$.

$\mathcal{P}$ holds the whole sharings $\{[\boldsymbol{w}_i]\}_{i \in [1,m]}$, $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i \in [1,M]}$ and $\{[\boldsymbol{y}'_i]\}_{i \in [1,\ell]}$.

Figure 6: **PSS-based strong NIMVZK in the $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$-hybrid model and random oracle model.**

Protocol $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$, continued

Procedure for Fiat-Shamir: All verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ and $\mathcal{P}$ execute as follows:

1. For $i \in [1, n]$, $\mathcal{P}$ samples $r_i \leftarrow \{0,1\}^\lambda$ and computes
$$com_i := \mathsf{H}_1(\hat{w}^i_1, \ldots, \hat{w}^i_m, \hat{z}^i_1, \ldots, \hat{z}^i_M, \hat{y}^i_1, \ldots, \hat{y}^i_\ell, r_i).$$
Then, $\mathcal{P}$ broadcasts $(com_1, \ldots, com_n)$ to all verifiers, and sends r_i to every verifier $\mathcal{V}_i$ over a private channel.
2. Each verifier $\mathcal{V}_i$ checks that $com_i = \mathsf{H}_1(\hat{w}^i_1, \ldots, \hat{w}^i_m, \hat{z}^i_1, \ldots, \hat{z}^i_M, \hat{y}^i_1, \ldots, \hat{y}^i_\ell, r_i)$, and aborts if the check fails.
3. $\mathcal{P}$ and all verifiers compute $\chi := \mathsf{H}_2(com_1, \ldots, com_n) \in \mathbb{K}$.

Verification of multiplication tuples: $\mathcal{P}$ and all verifiers execute as follows:

1. $\mathcal{P}$ and all verifiers respectively compute the secrets and the shares of $[\tilde{\boldsymbol{x}}_i] = \chi^{i-1} \cdot [\boldsymbol{x}_i]$, $[\tilde{\boldsymbol{y}}_i] = [\boldsymbol{y}_i]$ for $i \in [1, M]$ and $[\tilde{\boldsymbol{z}}] := \sum_{i \in [1,M]} \chi^{i-1} \cdot [\boldsymbol{z}_i]$.
2. The verifiers and $\mathcal{P}$ call functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ on packed inner-product tuple $(([\tilde{\boldsymbol{x}}_1], \ldots, [\tilde{\boldsymbol{x}}_M]), ([\tilde{\boldsymbol{y}}_1], \ldots, [\tilde{\boldsymbol{y}}_M]), [\tilde{\boldsymbol{z}}])$ to check that $\tilde{\boldsymbol{z}} = \sum_{h \in [1,M]} \tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h$. If the verifiers receive abort from $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, then they abort.

Verification of consistency of wire tuples: For all $i, j \in [1, k]$, $\mathcal{P}$ and all verifiers initiate an empty list $L(i, j)$. Then, from $h = 1$ to N, they insert $([\boldsymbol{x}_h], [\boldsymbol{y}_h], i_h, j_h)$ into the list $L(i_h, j_h)$. For each $i, j \in [1, k]$, $\mathcal{P}$ and all verifiers check the consistency of the wire tuples in $L(i, j)$ as follows:

1. Let N' be the size of $L(i, j)$. Let $([\boldsymbol{a}_1], [\boldsymbol{b}_1], i, j), \ldots, ([\boldsymbol{a}_{N'}], [\boldsymbol{b}_{N'}], i, j)$ denote the wire tuples in $L(i, j)$.
2. $\mathcal{P}$ samples $\boldsymbol{a}_0, \boldsymbol{b}_0 \leftarrow \mathbb{K}^k$ with $a_{0,i} = b_{0,j}$, and also picks $\boldsymbol{r}, \boldsymbol{r}' \leftarrow \mathbb{K}^k$. Then $\mathcal{P}$ and all verifiers execute the following:
   (a) $\mathcal{P}$ runs $[\boldsymbol{r}] \leftarrow \mathsf{Share}(\boldsymbol{r})$ and $[\boldsymbol{r}'] \leftarrow \mathsf{Share}(\boldsymbol{r}')$, which distributes the shares to all verifiers.
   (b) $\mathcal{P}$ computes $\boldsymbol{u} := \boldsymbol{a}_0 + \boldsymbol{r}$ and $\boldsymbol{u}' := \boldsymbol{b}_0 + \boldsymbol{r}'$, and then broadcasts $(\boldsymbol{u}, \boldsymbol{u}')$ to all verifiers.
   (c) The verifiers compute $[\boldsymbol{a}_0] := \boldsymbol{u} - [\boldsymbol{r}]$ and $[\boldsymbol{b}_0] := \boldsymbol{u}' - [\boldsymbol{r}']$.
3. All verifiers compute $\alpha := \mathsf{H}_2(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j) \in \mathbb{K}$.
4. All verifiers compute $[\boldsymbol{a}] := \sum_{h=0}^{N'} \alpha^h \cdot [\boldsymbol{a}_h]$ and $[\boldsymbol{b}] := \sum_{h=0}^{N'} \alpha^h \cdot [\boldsymbol{b}_h]$.
5. All verifiers run $\boldsymbol{a} \leftarrow \mathsf{Open}([\boldsymbol{a}])$ and $\boldsymbol{b} \leftarrow \mathsf{Open}([\boldsymbol{b}])$. If abort is output for the Open procedure, the verifiers abort. Otherwise, they check that $a_i = b_j$, and abort if the check fails.

Verification of circuit output: Let $[\boldsymbol{z}]$ be the sharing on the group of k circuit-output wires including the single actual circuit-output wire. All verifiers execute $\boldsymbol{z} \leftarrow \mathsf{Open}([\boldsymbol{z}])$, and abort if receiving abort from the Open procedure. If $\boldsymbol{z} = 0^k$, then the verifiers output true, otherwise they output false.

Figure 7: **PSS-based strong NIMVZK in the $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$-hybrid model and random oracle model, continued.**

realizes functionality $\mathcal{F}_{\mathsf{mvzk}}$ with soundness error at most $\frac{Q_1 n+(Q_2+1)(M+N)}{2^\lambda}$ in the $\mathcal{F}_{\mathsf{verifyprod}}^{\mathsf{pss}}$-hybrid model in the presence of a malicious adversary corrupting up to a prover and $t = d - k + 1$ verifiers, where degree-d packed sharings are used in protocol $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$, each sharing packs k secrets, and Q_1 and Q_2 are the number of queries to H_1 and H_2 respectively.

The proof of Theorem 3 can be found in Appendix F.

Optimizations. We can further optimize the protocol $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$ shown in Figures 6 and 7. Specifically, similar to the protocol $\Pi_{\mathsf{snimvzk}}^{\mathsf{fs}}$ described in Section 5 based on Shamir secret sharing, we can use the PRG and random seeds to reduce the communication of $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$ by t elements per sharing generation and improve the computational efficiency of generating and verifying t commitments. We can also use $\mathsf{H}_2(com_1, \ldots, com_n)$ to generate $\chi_1, \ldots, \chi_M \in \mathbb{K}$ instead of $1, \ldots, \chi^{M-1}$, and adopt $\mathsf{H}_2(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ to compute $\alpha_1, \ldots, \alpha_{N'}$ instead of $\alpha, \ldots, \alpha^{N'}$. In this way, the soundness error of protocol $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$ can be reduced to $\frac{Q_1 n+3(Q_2+1)}{2^\lambda}$.

Streaming strong NIMVZK proof. Protocol $\Pi_{\mathsf{snimvzk}}^{\mathsf{pss}}$ shown in Figures 6 and 7 are *streamable*, i.e., the circuit can be proved on-the-fly. The prover $\mathcal{P}$ can prove a batch of addition and multiplication gates each time, and stores the secrets that will be used as the input wire values in the next batches of gates.

For each batch of M multiplication gates, the verifiers need to check the correctness of a packed inner-product tuple $(([\tilde{\boldsymbol{x}}_1], \ldots, [\tilde{\boldsymbol{x}}_M]), ([\tilde{\boldsymbol{y}}_1], \ldots, [\tilde{\boldsymbol{y}}_M]), [\tilde{\boldsymbol{z}}])$. The verifiers can use the verification protocol shown in Section 6.2 to compress the packed inner-product tuple into a packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ with $m = M/2^s$ for some integer $s \geq 1$. For another batch of M multiplication gates, the verifiers can do the same operations and obtain a packed inner-product tuple $(([\boldsymbol{x}'_1], \ldots, [\boldsymbol{x}'_m]), ([\boldsymbol{y}'_1], \ldots, [\boldsymbol{y}'_m]), [\boldsymbol{z}'])$. Then, all verifiers can compress the two resulting packed inner-product tuples into a new inner-product tuple $(([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_m]), ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_m]), [\boldsymbol{c}])$. In this way, the verifiers can always store an inner-product tuple with a small dimension. When the whole circuit has been evaluated, the verifiers can run the verification protocol (as described in Section 6.2) to compress the packed inner-product tuple $(([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_m]), ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_m]), [\boldsymbol{c}])$ stored in memory into a multiplication tuple $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$, where note that $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ is also randomized with a random packed multiplication tuple generated by the prover. Then the verifiers run the one-round Open procedure to check correctness of $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$. By the above approach, the verifiers can communicate *only one round* for verifying all multiplication gates in the whole circuit.

For each batch of N wire tuples, the verifiers need to check their correctness. That is, for each $i, j \in [1, k]$, they will verify correctness of the wire tuples $([\boldsymbol{a}_1], [\boldsymbol{b}_1], i, j), \ldots, ([\boldsymbol{a}_{N'}], [\boldsymbol{b}_{N'}], i, j)$ in $L(i, j)$. By running the verification protocol shown in Figure 7, the verifiers can compress the wire tuples in $L(i, j)$ into a wire tuple $([\boldsymbol{a}], [\boldsymbol{b}], i, j)$. For another batch of N wire tuples, the verifiers need to check correctness of wire tuples $([\boldsymbol{a}'_1], [\boldsymbol{b}'_1], i, j), \ldots, ([\boldsymbol{a}'_{N''}], [\boldsymbol{b}'_{N''}], i, j)$ in $L(i, j)$ for each $i, j \in [1, k]$. Then, for each $i, j \in [1, k]$, the verifiers can run the verification protocol to compress $\{([\boldsymbol{a}'_h], [\boldsymbol{b}'_h], i, j)\}_{h \in [1, N'']}$ along with $([\boldsymbol{a}], [\boldsymbol{b}], i, j)$ into a new wire tuple $([\boldsymbol{a}'], [\boldsymbol{b}'], i, j)$. When the whole circuit has been evaluated, the verifiers open the k^2 wire tuples stored in memory to check their correctness. In this way, the verifiers can communicate only one round for all batches of wire tuples. Note that the Open procedure for verifying k^2 wire tuples can be executed in parallel with that for verifying an inner-product tuple. Therefore, only one-round communication is needed for the verifiers in total. Besides, only the resulting wire tuple that compresses the *final* batch of wire tuples needs to be randomized with a random wire tuple generated by the prover $\mathcal{P}$. Therefore, streaming the strong NIMVZK proof does *not* increase the communication.

Overall, using the above approach, we can prove a very large circuit when keeping the memory costs of the prover and verifiers *small*.

Protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$

Inputs: Prover $\mathcal{P}$ and all verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ respectively hold the secrets and shares of a packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_M]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_M]), [\boldsymbol{z}])$. $\mathcal{P}$ and all verifiers also hold a public value χ, which is determined after these secrets have been defined. The verifiers will check that $\boldsymbol{z} = \sum_{h \in [1,M]} \boldsymbol{x}_h * \boldsymbol{y}_h \in \mathbb{K}^k$. Let $\mathsf{H}_2 : \{0,1\}^* \to \mathbb{K}$ be a random oracle where $|\mathbb{K}| \geq 2^{\lambda}$.

- **Dimension-reduction:** Let m denote the dimension of the packed inner-product tuple in the current iteration, where m is initialized as M. Let γ be the public value in the current iteration, which is initialized as χ.
  While $m > 2$, $\mathcal{P}$ and all verifiers do the following:

  1. Let $\ell = m/2$. $\mathcal{P}$ and all verifiers define $([\boldsymbol{a}_{1,1}], \ldots, [\boldsymbol{a}_{1,\ell}]) := ([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_\ell])$ and $([\boldsymbol{a}_{2,1}], \ldots, [\boldsymbol{a}_{2,\ell}]) = ([\boldsymbol{x}_{\ell+1}], \ldots, [\boldsymbol{x}_m])$, where $\mathcal{P}$ holds all the secrets and the verifiers hold the shares. Similarly, they define $([\boldsymbol{b}_{1,1}], \ldots, [\boldsymbol{b}_{1,\ell}]) := ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_\ell])$ and $([\boldsymbol{b}_{2,1}], \ldots, [\boldsymbol{b}_{2,\ell}]) = ([\boldsymbol{y}_{\ell+1}], \ldots, [\boldsymbol{y}_m])$.
  2. $\mathcal{P}$ and all verifiers execute the sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ (shown in Figure 9) on $([\boldsymbol{a}_{1,1}], \ldots, [\boldsymbol{a}_{1,\ell}])$ and $([\boldsymbol{b}_{1,1}], \ldots, [\boldsymbol{b}_{1,\ell}])$ to compute the whole sharing $[\boldsymbol{c}_1] = [\sum_{h \in [1,\ell]} \boldsymbol{a}_{1,h} * \boldsymbol{b}_{1,h}]$, where the secrets and the shares are output to $\mathcal{P}$ and the verifiers respectively. $\mathcal{P}$ and all verifiers also obtain $\boldsymbol{u} \in \mathbb{K}^k$.
  3. $\mathcal{P}$ and all verifiers compute $[\boldsymbol{c}_2] := [\boldsymbol{z}] - [\boldsymbol{c}_1]$, where $\mathcal{P}$ obtains $\boldsymbol{c}_2$ and the verifiers get the shares of $[\boldsymbol{c}_2]$.
  4. $\mathcal{P}$ and all verifiers execute sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ on $([\boldsymbol{a}_{i,1}], \ldots, [\boldsymbol{a}_{i,\ell}])$, $([\boldsymbol{b}_{i,1}], \ldots, [\boldsymbol{b}_{i,\ell}])$ and $[\boldsymbol{c}_i]$ for $i \in [1,2]$ and $(\gamma, \boldsymbol{u})$, where $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ is described in Figure 10.
  5. $\mathcal{P}$ and all verifiers update γ as the public element α output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$, and also set $m := m/2$. Then, $\mathcal{P}$ and the verifiers use the whole output sharings $(([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\ell]), ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\ell]), [\boldsymbol{c}])$ from $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ to update $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$.

- **Randomization:** Let $\gamma \in \mathbb{K}$ along with the inner-product tuple $(([\boldsymbol{x}_1], [\boldsymbol{x}_2]), ([\boldsymbol{y}_1], [\boldsymbol{y}_2]), [\boldsymbol{z}])$ be the final output from the previous phase. $\mathcal{P}$ and all verifiers execute the following procedure:

  1. $\mathcal{P}$ samples $\boldsymbol{x}_0, \boldsymbol{y}_0 \leftarrow \mathbb{K}^k$, and then runs $[\boldsymbol{x}_0] \leftarrow \mathsf{Share}(\boldsymbol{x}_0)$ and $[\boldsymbol{y}_0] \leftarrow \mathsf{Share}(\boldsymbol{y}_0)$, which distribute the shares to all verifiers.
  2. For $i \in [0,1]$, $\mathcal{P}$ and all verifiers execute the sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ on $([\boldsymbol{x}_i], [\boldsymbol{y}_i])$ to compute the whole sharing $[\boldsymbol{z}_i] = [\boldsymbol{x}_i * \boldsymbol{y}_i]$. Additionally, $\mathcal{P}$ and the verifiers also obtain $\boldsymbol{u}_0, \boldsymbol{u}_1 \in \mathbb{K}^k$.
  3. $\mathcal{P}$ and all verifiers compute $[\boldsymbol{z}_2] := [\boldsymbol{z}] - [\boldsymbol{z}_1]$, where $\mathcal{P}$ obtains $\boldsymbol{z}_2$ and the verifiers get the shares of $[\boldsymbol{z}_2]$.
  4. $\mathcal{P}$ and all verifiers execute sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ on $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i \in [0,2]}$ and $(\gamma, \boldsymbol{u}_0, \boldsymbol{u}_1)$. Then, all verifiers obtain the output $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$.
  5. All verifiers run $\boldsymbol{v} \leftarrow \mathsf{Open}([\boldsymbol{v}])$ for each $\boldsymbol{v} \in \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. If abort is received during the Open procedure, then the verifiers abort. Then, the verifiers check that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. If the check fails, the verifiers output abort. Otherwise, they output accept.

Figure 8: **One-round ZK verification protocol for packed inner-product tuples.**

Protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$

Inputs: Prover $\mathcal{P}$ and verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ respectively hold the secrets and the shares of packed sharings $([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_\ell])$ and $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_\ell])$ over a field $\mathbb{K}$.

Protocol execution: $\mathcal{P}$ and all verifiers execute as follows:

1. $\mathcal{P}$ samples $\boldsymbol{r} \leftarrow \mathbb{K}^k$, and then runs $[\boldsymbol{r}] \leftarrow \mathsf{Share}(\boldsymbol{r})$ that distributes the shares to all verifiers.
2. $\mathcal{P}$ computes $\boldsymbol{z} := \sum_{h \in [1,\ell]} \boldsymbol{x}_h * \boldsymbol{y}_h \in \mathbb{K}^k$, and then broadcasts $\boldsymbol{u} := \boldsymbol{z} + \boldsymbol{r} \in \mathbb{K}^k$ to all verifiers.
3. The verifiers compute $[\boldsymbol{z}] := \boldsymbol{u} - [\boldsymbol{r}]$. Then, $\mathcal{P}$ and the verifiers respectively output the secrets and shares of $[\boldsymbol{z}]$, and also output $\boldsymbol{u} \in \mathbb{K}^k$.

Figure 9: **Non-interactive inner-product protocol for packed sharings secure up to additive errors.**

Protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$

Inputs: Let m be the number of packed inner-product tuples, and ℓ be the dimension of each packed inner-product tuple. For $i \in [1, m]$, prover $\mathcal{P}$ and all verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ respectively hold the secrets and shares of packed inner-product tuple $(([\boldsymbol{x}_{i,1}], \ldots, [\boldsymbol{x}_{i,\ell}]), ([\boldsymbol{y}_{i,1}], \ldots, [\boldsymbol{y}_{i,\ell}]), [\boldsymbol{z}_i])$. $\mathcal{P}$ and all verifiers also input $\gamma \in \mathbb{K}$ and $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d \in \mathbb{K}^k$. Let $\mathsf{H}_2 : \{0,1\}^* \rightarrow \mathbb{K}$ be a random oracle.

Protocol execution: $\mathcal{P}$ and all verifiers execute as follows:

1. For each $j \in [1, \ell]$, $\mathcal{P}$ computes vectors of degree-$(m-1)$ polynomials $\boldsymbol{f}_j(\cdot)$ and $\boldsymbol{g}_j(\cdot)$, such that $\boldsymbol{f}_j(i) = \boldsymbol{x}_{i,j}$ and $\boldsymbol{g}_j(i) = \boldsymbol{y}_{i,j}$ for all $i \in [1, m]$.
2. For $j \in [1, \ell]$, all verifiers locally compute $[\boldsymbol{f}_j(\cdot)]$ and $[\boldsymbol{g}_j(\cdot)]$ using their shares of $\{[\boldsymbol{x}_{i,j}]\}_{i \in [1,m]}$ and $\{[\boldsymbol{y}_{i,j}]\}_{i \in [1,m]}$ respectively.
3. For $i \in [m+1, 2m-1]$, $\mathcal{P}$ and all verifiers respectively compute the secrets and shares of packed sharings $[\boldsymbol{f}_j(i)]$ and $[\boldsymbol{g}_j(i)]$ for $j \in [1, \ell]$. Then, for $i \in [m+1, 2m-1]$, they execute the sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ (shown in Figure 9) on $([\boldsymbol{f}_1(i)], \ldots, [\boldsymbol{f}_\ell(i)])$ and $([\boldsymbol{g}_1(i)], \ldots, [\boldsymbol{g}_\ell(i)])$ to compute the whole sharing $[\boldsymbol{z}_i] = [\sum_{j \in [1,\ell]} \boldsymbol{f}_j(i) * \boldsymbol{g}_j(i)]$, where $\mathcal{P}$ obtains $\boldsymbol{z}_i$ and the verifiers get the shares of $[\boldsymbol{z}_i]$. Besides, $\mathcal{P}$ and the verifiers obtain $\boldsymbol{u}_{m+1}, \ldots, \boldsymbol{u}_{2m-1} \in \mathbb{K}^k$.
4. $\mathcal{P}$ and all verifiers locally compute the whole sharing $[\boldsymbol{h}(\cdot)]$ from $[\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_{2m-1}]$, such that $\boldsymbol{h}(\cdot)$ is a vector of degree-$2(m-1)$ polynomial and $\boldsymbol{h}(i) = \boldsymbol{z}_i$ for $i \in [1, 2m-1]$.
5. $\mathcal{P}$ and all verifiers compute $\alpha := \mathsf{H}_2(\gamma, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, \boldsymbol{u}_{m+1}, \ldots, \boldsymbol{u}_{2m-1}) \in \mathbb{K}$. If $\alpha \in [1, m]$, the verifiers abort.
6. $\mathcal{P}$ and all verifiers output the secrets and shares of $([\boldsymbol{f}_1(\alpha)], \ldots, [\boldsymbol{f}_\ell(\alpha)])$, $([\boldsymbol{g}_1(\alpha)], \ldots, [\boldsymbol{g}_\ell(\alpha)])$ and $[\boldsymbol{h}(\alpha)]$ respectively, and also output the element α.

Figure 10: **Protocol for compressing packed inner-product tuples.**

6.2 Strong NIMVZK Proof for Packed Inner-Product Tuples

Below, we present a strongly non-interactive MVZK protocol with *logarithmic* communication complexity to verify packed inner-product tuples, which is inspired by the technique by Goyal et al. [GPS21] for MPC that is in turn built on the techniques [BBC$^+$19, GS20, GSZ20]. While the MPC protocol [GPS21] requires logarithmic rounds to check correctness of packed inner-product tuples, our strong NIMVZK protocol needs only one round between verifiers. Furthermore, our protocol reduces the communication overhead for verification by making the prover generate the random sharings and messages associated with secrets, compared to the verification of packed inner-product tuples directly using the MPC protocol [GPS21].

Our PSS-based strong NIMVZK protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ for verifying a packed inner-product tuple is described in Figure 8. This protocol invokes two sub-protocols $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ and $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ that are described in Figures 9 and 10 respectively, where $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ is used to generate the inner product of two vectors and $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ is used to compress two packed inner-product tuples into a single tuple. In the dimension-reduction and randomization phases of this protocol, we adapt the approach by Baum et al. [BMRS21] used in the DVZK setting to non-interactively generate a challenge α used in sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ based on the Fiat-Shamir transform. Based on the *round-by-round soundness* [CCH+19, BMRS21], we can prove that the soundness error of our protocol is negligible (see Theorem 4). In the dimension-reduction phase of $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$, we always assume that the dimension m of the packed inner-product tuple is the multiple of 2 for each iteration. If not, we can pad the dummy zero sharing $[\mathbf{0}]$ into the packed inner-product tuple to satisfy the requirement, where $[\mathbf{0}]$ can be locally computed by all verifiers.

In the protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ shown in Figure 8, we assume that $\mathcal{P}$ and all verifiers input a public challenge χ, which is determined after the secrets packed in the input inner-product tuple $(([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_M]), ([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_M]), [\boldsymbol{z}])$ have been defined. In particular, χ can be defined as $\mathsf{H}_2(com_1, \dots, com_n)$ as shown in Figure 7. When using $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ to realize functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, $\chi \in \mathbb{K}$ can be generated by $\mathcal{P}$ and all verifiers in the main NIMVZK protocol shown in Figures 6 and 7. For the sake of simplicity, we ignore the case that the adversary (who corrupts $\mathcal{P}$) did not make a query to obtain χ but made a query to get a challenge $\alpha = \mathsf{H}_2(\chi, \cdots)$ used in protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$, which occurs with probability at most $\frac{1}{|\mathbb{K}|} \leq \frac{1}{2^\lambda}$. When the adversary makes a query $(com_1, \dots, com_n)$ to random oracle H_2 and obtains χ, the challenge χ is determined after the secrets stored in the input packed tuple have been defined, except with probability at most $\frac{Q_1 n}{2^\lambda}$ following the proof of Theorem 3.

Sub-protocol for computing inner product. Sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ shown in Figure 9 is used to compute the inner product of two vectors $([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_\ell])$ and $([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_\ell])$. The prover now knows all the challenges due to the use of the Fiat-Shamir transform, and thus holds all the secrets involved in the verification procedure. Thus, the prover can directly distribute the shares of $[\boldsymbol{z}]$ with $\boldsymbol{z} = \sum_{h \in [1,\ell]} \boldsymbol{x}_h * \boldsymbol{y}_h$ to all verifiers. To support Fiat-Shamir, the verifiers need to know public messages instead of secret shares. Therefore, we first let the prover generate a random packed sharing $[\boldsymbol{r}]$, and then make it broadcast the *public* difference $\boldsymbol{u} = \boldsymbol{z} + \boldsymbol{r}$ to all verifiers. The prover and verifiers also need to output the message $\boldsymbol{u}$, which will be used in the Fiat-Shamir transform of the main verification protocol. When the prover is malicious, it can introduce an additive error to the sharing $[\boldsymbol{z}]$ output by the verifiers, which is harmless when integrating $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ into the main protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$.

Sub-protocol for compression. Sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ shown in Figure 10 is used to compress m packed inner-product tuples into a single packed inner-product tuple. In particular, this protocol invokes the sub-protocol $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ instead of calling an inner-product functionality, which seems necessary to support Fiat-Shamir, where the messages related to the secrets need to be used as the input of a random oracle H_2. For every protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$, the prover and all verifiers generate a random challenge $\alpha \in \mathbb{K}$ using the Fiat-Shamir transform. To realize *non-interactively* recursive compression in the main verification protocol, the prover and verifiers also input the public challenge γ from the previous iteration and the public messages produced in the current iteration.

Theorem 4. *Let $\mathsf{H}_2 : \{0,1\}^* \to \mathbb{K}$ be a random oracle. Protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ shown in Figure 8 securely realizes functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ with soundness error at most $\frac{4\lceil \log M \rceil + 5Q_2}{2^\lambda - 3}$ in the presence of a malicious adversary corrupting up to the prover and exactly t verifiers, where Q_2 is the number of queries to random oracle H_2.*

The proof of Theorem 4 is given in Appendix G.

Acknowledgements

Work of Kang Yang is supported by the National Natural Science Foundation of China (Grant Nos. 62102037, 61932019). Work of Xiao Wang is supported in part by DARPA under Contract No. HR001120C0087, NSF award #2016240, and research awards from Facebook and PlatON Network. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[Abe99] Masayuki Abe. Robust distributed multiplication without interaction. In *Advances in Cryptology—Crypto 1999*, volume 1666 of *LNCS*, pages 130–147. Springer, 1999.

[ACF02] Masayuki Abe, Ronald Cramer, and Serge Fehr. Non-interactive distributed-verifier proofs and proving relations among commitments. In *Advances in Cryptology—Asiacrypt 2002*, LNCS, pages 206–223. Springer, 2002.

[ACGJ19] Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Two round information-theoretic MPC with malicious security. In *Advances in Cryptology—Eurocrypt 2019, Part II*, volume 11477 of *LNCS*, pages 532–561. Springer, 2019.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM Conf. on Computer and Communications Security (CCS) 2017*, pages 2087–2104. ACM Press, 2017.

[BBB$^+$18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symp. Security and Privacy 2018*, pages 315–334. IEEE, 2018.

[BBC$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *Advances in Cryptology—Crypto 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, 2019.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology—Crypto 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, 2019.

[BBMH$^+$21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and z2k. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security – CCS'21*, pages 192–211. ACM, 2021.

[BCC$^+$16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology—Eurocrypt 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, 2016.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography*, pages 315–333. Springer Berlin Heidelberg, 2013.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology—Eurocrypt 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, 2019.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *9th Theory of Cryptography Conference—TCC 2016*, LNCS, pages 31–60. Springer, 2016.

[BD91] Mike Burmester and Yvo Desmedt. Broadcast interactive proofs (extended abstract). In *Advances in Cryptology—Eurocrypt 1991*, LNCS, pages 81–95. Springer, 1991.

[BFH+20] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 2025–2038. ACM Press, 2020.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 663–680. Springer, 2012.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Advances in Cryptology—Eurocrypt 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, 2020.

[BGIN20] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *Advances in Cryptology—Asiacrypt 2020, Part III*, LNCS, pages 244–276. Springer, 2020.

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In *Advances in Cryptology—Eurocrypt 2021, Part II*, LNCS, pages 663–693. Springer, 2021.

[BKZZ20] Foteini Baldimtsi, Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. Crowd verifiable zero-knowledge and end-to-end verifiable multiparty computation. In *Advances in Cryptology—Asiacrypt 2020, Part III*, LNCS, pages 717–748. Springer, 2020.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *Advances in Cryptology—Crypto 2021, Part IV*, LNCS, pages 92–122. Springer, 2021.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *5th Theory of Cryptography Conference—TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, 2008.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, January 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145. IEEE, 2001.

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In *51th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1082–1090. ACM Press, 2019.

[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)*, pages 259–282. USENIX Association, March 2017.

[CGH$^+$18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology—Crypto 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, 2018.

[CK21] Ran Canetti and Gabriel Kaptchuk. The broken promise of apple's announced forbidden-photo reporting system – and how to fix it. `https://www.bu.edu/riscs/2021/08/10/apple-csam/`, 2021.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances in Cryptology—Eurocrypt 2010*, LNCS, pages 445–465. Springer, 2010.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology—Crypto 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, 2007.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Advances in Cryptology—Crypto 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, 2017.

[FL19] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *ACM Conf. on Computer and Communications Security (CCS) 2019*, pages 1557–1571. ACM Press, 2019.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—Crypto 1986*, LNCS, pages 186–194. Springer, 1987.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 699–710. ACM Press, 1992.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Advances in Cryptology—Eurocrypt 2013*, LNCS, pages 626–645. Springer, 2013.

[GIOZ17] Juan A. Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In *Advances in Cryptology—Crypto 2017, Part I*, volume 10401 of *LNCS*, pages 420–446. Springer, 2017.

[GIP$^+$14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 495–504. ACM Press, 2014.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In *Advances in Cryptology—Crypto 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, 2015.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 113–122. ACM Press, 2008.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18(3):247–287, July 2005.

[GLO$^+$21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In *Advances in Cryptology—Crypto 2021, Part II*, LNCS, pages 244–274. Springer, 2021.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In *Advances in Cryptology—Crypto 2021, Part II*, LNCS, pages 275–304. Springer, 2021.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Advances in Cryptology—Asiacrypt 2010*, LNCS, pages 321–340. Springer, 2010.

[GS20] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134.

[GSY21] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In *Advances in Cryptology—Eurocrypt 2021, Part II*, LNCS, pages 694–723. Springer, 2021.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In *Advances in Cryptology—Crypto 2020, Part II*, LNCS, pages 618–646. Springer, 2020.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 21–30. ACM Press, 2007.

[KLR06] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. In *38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 109–118. ACM Press, 2006.

[LMs05] Matt Lepinski, Silvio Micali, and abhi shelat. Fair-zero knowledge. In *2nd Theory of Cryptography Conference—TCC 2005*, volume 3378 of *LNCS*, pages 245–263. Springer, 2005.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM Conf. on Computer and Communications Security (CCS) 2017*, pages 259–276. ACM Press, 2017.

[NV18] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *Intl. Conference on Applied Cryptography and Network Security (ACNS)*, LNCS, pages 321–339. Springer, 2018.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Advances in Cryptology—Crypto 2020, Part III*, LNCS, pages 704–737. Springer, 2020.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[WTs$^+$18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symp. Security and Privacy 2018*, pages 926–943. IEEE, 2018.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE Symp. Security and Privacy 2021*. IEEE, 2021.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM Conf. on Computer and Communications Security (CCS) 2021*. ACM Press, 2021.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE Symp. Security and Privacy 2020*, pages 859–876. IEEE, 2020.

A More Preliminaries

A.1 Security Model

We use the standard ideal/real paradigm [Can00, Gol04] to prove security of our NIMVZK protocols in the presence of a *malicious, static* adversary. In the *ideal-world* execution, the parties interact with a functionality $\mathcal{F}$, and some of them may be corrupted by an *ideal-world adversary* (a.k.a., *simulator*) $\mathcal{S}$. In the *real-world* execution, the parties interact with each other in an execution of protocol Π, and some of them may be corrupted by a *real-world adversary* $\mathcal{A}$ (that is often called an adversary for simplicity). Let $\mathsf{REAL}_{\Pi,\mathcal{A}}(\kappa, z)$ denote the output of the honest parties and $\mathcal{A}$ in a real-world execution of protocol Π with a security parameter κ and an auxiliary input z for $\mathcal{A}$. Let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}(\kappa, z)$ be the output of the honest parties and $\mathcal{S}$ in an ideal-world execution with the functionality $\mathcal{F}$, a security parameter κ and an auxiliary input z to $\mathcal{S}$. Let $\mathsf{REAL}_{\Pi,\mathcal{A}}$ be the ensemble $\{\mathsf{REAL}_{\Pi,\mathcal{A}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$, and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}$ be the ensemble $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$.

Definition 1. *We say that a protocol* Π securely realizes *an ideal functionality* $\mathcal{F}$*, if for any adversary* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that*

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}} \approx \mathsf{REAL}_{\Pi,\mathcal{A}},$$

where $\approx$ *denotes the statistical (or computational) distinguishability of two distributions.*

In the information-theoretic setting, the real-world adversary $\mathcal{A}$ are computationally unbounded (where the ideal-world adversary $\mathcal{S}$ runs in time that is polynomial in the running time of $\mathcal{A}$), and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}$ is *statistically* distinguishable from $\mathsf{REAL}_{\Pi,\mathcal{A}}$ which is denoted by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}} \overset{\text{s}}{\approx} \mathsf{REAL}_{\Pi,\mathcal{A}}$. In the computational setting, the adversaries in the two worlds run in non-uniform probabilistic polynomial time (PPT), and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}} \overset{\text{c}}{\approx} \mathsf{REAL}_{\Pi,\mathcal{A}}$ where $\overset{\text{c}}{\approx}$ denotes the computational distinguishability of two distributions. We prove security of our NIMVZK protocols in the $\mathcal{G}$-hybrid model, where the parties execute a protocol with real messages and also have access to a sub-functionality $\mathcal{G}$. In [KLR06, Theorem 5], it has been shown that any protocol, that is proven secure in the stand-alone model with a black-box straight-line simulator and start synchronization (i.e., the inputs of all parties are fixed before the protocol execution starts), is also secure under the universal composability (UC) framework [Can01]. All our protocols satisfy these conditions, and thus are also UC-secure. This allows us to run the protocols in parallel and concurrently. In all our proofs of security, we will use $\mathcal{H}$ and $\mathcal{C}$ to denote the sets of honest verifiers and corrupted verifiers, respectively.

A.2 Two Instantiations of Linear Secret Sharings

Instantiation 1: Shamir secret sharing [Sha79]. The standard Shamir secret sharing scheme allows any $t < n/2$. Following prior MPC work, we always set $n = 2t + 1$ for the sake of simplicity. Let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$ be n predetermined distinct non-zero elements. The procedures for Shamir secret sharing over a field $\mathbb{F}$ with $|\mathbb{F}| > n$ are defined as follows:

- $[x] \leftarrow \mathsf{Share}(x)$: On input a secret $x \in \mathbb{F}$, a dealer $\mathcal{P}$ samples $x^1, \ldots, x^t \leftarrow \mathbb{F}$, and then uses Lagrange interpolation to compute a polynomial $f(X) \in \mathbb{F}[X]$ of degree at most t, such that $f(0) = x$ and $f(\alpha_i) = x^i$ for $i \in [1, t]$. For each $i \in [1, n]$, $\mathcal{P}$ sends $x^i = f(\alpha_i)$ to $\mathcal{V}_i$ over a private channel.

  In the computational setting, the communication could be further reduced using a *pseudo-random generator* (PRG).Since t out of n shares are uniformly random, they can be generated by the pre-agreed seeds, which can be reused across multiple different sharings. This reduces the communication by a factor of 2.

- $x \leftarrow \mathsf{Open}([x])$: On input a sharing $[x]$, every party $\mathcal{V}_i$ sends its share x^i to all other parties $\{\mathcal{V}_j\}_{j \in [1,n], j \neq i}$. After obtaining n shares $x^1, \ldots, x^n$, every party $\mathcal{V}_i$ *reconstructs* the secret x as follows:

  1. Use the first $t+1$ shares to compute the unique degree-t polynomial $f(\cdot)$ using Lagrange interpolation.
  2. Check $x^i = f(\alpha_i)$ for all $i \in [t+2, n]$. If the check fails, then output abort; otherwise output $x = f(0)$.

- For a constant $v \in \mathbb{F}$ that is known by all parties, it is convenient to *locally* transform it to a Shamir secret sharing by defining a constant polynomial $f(X) = v$ and the share of every party $\mathcal{V}_i$ to be $f(\alpha_i) = v$.

To explicitly express the degree t, we sometimes denote by $\mathsf{Share}_t(\cdot)$ the Share procedure. For degree $2t < n$, the procedure $\mathsf{Share}_{2t}(\cdot)$ can be defined similarly.

Instantiation 2: Packed secret sharing [FY92]. The packed secret sharing scheme is a generalization of the standard Shamir secret sharing scheme described as above. Let k be the number of secrets that are packed in a single sharing. Similar to the recent MPC work [GSY21, GPS21], we adopt $n = 2d + 1$ for simplicity, where $d = t + k - 1$ is the degree of polynomials. For example, we can choose to adopt $t = \lfloor (n-1)/3 \rfloor$ and $k = \lfloor (n-1)/6 \rfloor + 1$. When setting $k = 1$, packed secret sharing becomes Shamir secret sharing. In addition to $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, we also need k more predetermined distinct elements $\beta_1, \ldots, \beta_k \in \mathbb{F}$, which are different from $\alpha_1, \ldots, \alpha_n$. The procedures for packed secret sharing over a field $\mathbb{F}$ with $|\mathbb{F}| \geq n + k$ are defined as follows:

- $[\boldsymbol{x}] \leftarrow \mathsf{Share}(\boldsymbol{x})$: On input a vector of secrets $\boldsymbol{x} \in \mathbb{F}^k$, a dealer $\mathcal{P}$ samples $s^1, \ldots, s^t \leftarrow \mathbb{F}$, and then uses the interpolation approach to compute a polynomial $f(X) \in \mathbb{F}[X]$ of degree at most $d = t + k - 1$, such that $f(\beta_i) = x_i$ for $i \in [1, k]$ and $f(\alpha_i) = s^i$ for $i \in [1, t]$. For $i \in [1, n]$, $\mathcal{P}$ sends $s^i = f(\alpha_i)$ to $\mathcal{V}_i$ over a private channel.

  Similar to Shamir secret sharing, the communication complexity can also be reduced using a PRG in the computational setting, where the shares for $\mathcal{V}_1, \ldots, \mathcal{V}_t$ can be computed with the random seeds and PRG.

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

This functionality runs with n parties $P_1, \ldots, P_n$ as follows:

- Upon receiving $(\mathsf{coin}, \mathbb{K})$ from all parties where $\mathbb{K}$ is a field, sample $r \leftarrow \mathbb{K}$ and send r to all parties.

Figure 11: **Coin-tossing functionality.**

- $\boldsymbol{x} \leftarrow \mathsf{Open}([\boldsymbol{x}])$: Given a packed sharing $[\boldsymbol{x}]$, every party $\mathcal{V}_i$ sends its share s^i to all other parties $\{\mathcal{V}_j\}_{j\in[1,n],j\neq i}$. After obtaining n shares $s^1, \ldots, s^n$, every party $\mathcal{V}_i$ reconstructs the vector $\boldsymbol{x}$ as follows:
  1. Use the first $d+1$ shares to compute the degree-d polynomial $f(\cdot)$ based on Lagrange interpolation.
  2. Check that $s^i = f(\alpha_i)$ for all $i \in [d+2, n]$. If the check fails, then output abort; otherwise, output a vector $\boldsymbol{x} \in \mathbb{F}^k$ such that $x_i = f(\beta_i)$ for $i \in [1, k]$.
- For a constant vector $\boldsymbol{v} \in \mathbb{F}^k$ that is known by all parties, it is convenient to transform it to a packed secret sharing *without any communication* as follows:
  1. All parties define a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree $k-1$ such that $f(\beta_i) = v_i$ for each $i \in [1, k]$.
  2. Every party $\mathcal{V}_i$ defines $f(\alpha_i)$ as its share.

It is well known that the secret-sharing schemes described as above are linear and allows local linear combination. Following the prior work such as [LN17, BGIN20, GPS21], the Open procedure for Shamir or packed secret sharings guarantees that all honest parties can output the same correct values, when there are at most $t < n/2$ (resp., $t \le n/2 - k$) malicious parties for Shamir (resp., packed) secret sharing.

A.3 Coin-Tossing Functionality

In the *information-theoretic* setting, our NIMVZK protocol will use a coin-tossing functionality shown in Figure 11 to generate a public random element in a field $\mathbb{K}$. This functionality can be efficiently realized using the standard protocol in the honest-majority setting, which has been used in previous MPC work such as [BTH08, LN17, CGH$^+$18, FL19, GSZ20, BGIN20]. The coin-tossing protocol works as follows:

1. Every party shares a random element to the other parties.
2. Then, the parties transform n sharings into $n-t$ random sharings using the Vandermonde matrix.
3. Finally, the parties run the Open procedure to obtain a public random element from a random sharing.

B Proof of Lemma 1

Lemma 2 (Lemma 1, restated)**.** *Let Π be an MVZK protocol proving the satisfiability of a circuit C for $n \ge 2t+1$ verifiers. Then, if protocol Π securely realizes $\mathcal{F}_{\mathsf{mvzk}}$ in the presence of any malicious adversary corrupting exactly t verifiers, then Π securely realizes $\mathcal{F}_{\mathsf{mvzk}}$ against any malicious adversary corrupting at most t verifiers.*

Proof. Let $\mathcal{P}$ be the prover and $\mathcal{V}_1, \ldots, \mathcal{V}_n$ be the set of n verifiers in a protocol execution of Π. Let $\mathcal{A}$ be a malicious adversary who corrupts a set $\mathcal{C}$ of verifiers such that $|\mathcal{C}| < t$. (If $|\mathcal{C}| = t$, then this proof is straightforward and thus omitted.) Fix $\mathcal{C}^*$ to be some minimum set of verifiers such that $|\mathcal{C}^* \cup \mathcal{C}| = t$. Consider the adversary $\mathcal{A}^*$ who controls the verifiers in $\mathcal{C}^* \cup \mathcal{C}$. If $\mathcal{A}$ corrupts the prover $\mathcal{P}$, then $\mathcal{A}^*$ also corrupts $\mathcal{P}$. Adversary $\mathcal{A}^*$ behaves as follows:

1. $\mathcal{A}^*$ honestly simulates the behavior of the verifiers in $\mathcal{C}^*$. In particular, when a verifier in $\mathcal{C}^*$ intends to send a message m to another verifier in $\mathcal{C}$ or $\mathcal{P}$ (if $\mathcal{P}$ is corrupted), $\mathcal{A}^*$ *internally* sends m to $\mathcal{A}$.

2. $\mathcal{A}^*$ simulates $\mathcal{A}$ on the verifiers in $\mathcal{C}$ and the prover $\mathcal{P}$ (if $\mathcal{P}$ is corrupted). Specifically, when $\mathcal{A}$ wants to send a message m to a verifier in $\mathcal{C}^*$, $\mathcal{A}^*$ *internally* sends m to the verifier in $\mathcal{C}^*$.

3. If a random oracle H is involved in protocol Π, then whenever $\mathcal{A}$ makes a query to H, $\mathcal{A}^*$ makes the same query to H.

Note that $\mathcal{A}^*$ corrupts exactly t verifiers, and thus Π is secure against adversary $\mathcal{A}^*$. In particular, there exists a simulator $\mathcal{S}^*$ such that

$$\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mvzk}},\mathcal{S}^*,\mathcal{C}^*\cup\mathcal{C}} \approx \mathsf{REAL}_{\Pi,\mathcal{A}^*,\mathcal{C}^*\cup\mathcal{C}},$$

where the set of corrupted verifiers are explicitly written. Below, using $\mathcal{S}^*$ and $\mathcal{A}^*$, we construct a simulator $\mathcal{S}$ for adversary $\mathcal{A}$ as follows:

1. If the prover $\mathcal{P}$ is corrupted, $\mathcal{S}$ invokes $\mathcal{S}^*$ to extract a witness $\boldsymbol{w}$ from $\mathcal{A}$.

2. Whenever $\mathcal{S}^*$ calls functionality $\mathcal{F}_{\mathsf{mvzk}}$, $\mathcal{S}$ also calls $\mathcal{F}_{\mathsf{mvzk}}$. In the case that $\mathcal{P}$ is corrupted, if $\mathcal{S}^*$ wants to send the witness to $\mathcal{F}_{\mathsf{mvzk}}$, then $\mathcal{S}$ sends $\boldsymbol{w}$ to $\mathcal{F}_{\mathsf{mvzk}}$. After receiving a single-bit output from $\mathcal{F}_{\mathsf{mvzk}}$, $\mathcal{S}$ forwards it to $\mathcal{S}^*$. If $\mathcal{S}^*$ wants to send abort or $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{mvzk}}$, $\mathcal{S}$ forwards it to $\mathcal{F}_{\mathsf{mvzk}}$.

3. $\mathcal{S}$ invokes $\mathcal{S}^*$ to simulate the view of $\mathcal{A}$, where $\mathcal{S}$ simulates the verifiers in $\mathcal{C}^*$ by invoking $\mathcal{A}^*$ with the view simulated by $\mathcal{S}^*$.

4. If a random oracle H is involved in protocol Π, $\mathcal{S}$ responds the random-oracle queries by invoking $\mathcal{S}^*$.

In the following, we prove that

$$\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mvzk}},\mathcal{S},\mathcal{C}} \approx \mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{C}}.$$

Let $\mathsf{View}^{\mathcal{A}}_{\mathcal{C}}$ denote the view of $\mathcal{A}$ in the real protocol execution, and $\mathsf{View}^{\mathcal{S}}_{\mathcal{C}}$ be the view simulated by $\mathcal{S}$ for $\mathcal{A}$ in the ideal-world execution. Similarly, we can define $\mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}$ and $\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}}$. From $\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mvzk}},\mathcal{S}^*,\mathcal{C}^*\cup\mathcal{C}} \approx \mathsf{REAL}_{\Pi,\mathcal{A}^*,\mathcal{C}^*\cup\mathcal{C}}$, we have that

$$\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}} \approx \mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}.$$

We denote by $\mathsf{Msg}^{\mathcal{S}^*}_{\mathcal{C}^*}$ be the messages, which are sent by the verifiers in $\mathcal{C}^*$ to the verifiers in $\mathcal{C}$ and the prover $\mathcal{P}$ (if $\mathcal{P}$ is corrupted), in the step 3 of the above simulation of $\mathcal{S}$. Let $\mathsf{Msg}^{\mathcal{A}^*}_{\mathcal{C}^*}$ be the messages that are sent among the same parties during the step 1 inside $\mathcal{A}^*$ in the real protocol execution. Both of $\mathsf{Msg}^{\mathcal{S}^*}_{\mathcal{C}^*}$ and $\mathsf{Msg}^{\mathcal{A}^*}_{\mathcal{C}^*}$ are computed by $\mathcal{A}^*$, except that $\mathcal{A}^*$ is given the view simulated by $\mathcal{S}^*$ in the first one and the real view in the second one. From $\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}} \approx \mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}$, we easily have

$$\left(\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathsf{Msg}^{\mathcal{S}^*}_{\mathcal{C}^*}\right) \approx \left(\mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathsf{Msg}^{\mathcal{A}^*}_{\mathcal{C}^*}\right).$$

Let $\mathsf{FixView}(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}, \mathcal{C}^*, \mathsf{Msg}_{\mathcal{C}^*})$ be a function, which removes from $\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}$ the incoming messages to the verifiers in $\mathcal{C}^*$ and adds to $\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}$ the messages $\mathsf{Msg}_{\mathcal{C}^*}$ that are sent by the verifiers in $\mathcal{C}^*$ to the verifiers in $\mathcal{C}$ and the prover $\mathcal{P}$ (if $\mathcal{P}$ is corrupted), where $\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}$ is a view of the verifiers in $\mathcal{C}^*\cup\mathcal{C}$ and the prover $\mathcal{P}$ (if $\mathcal{P}$ is corrupted). In the real protocol execution, we note that $\mathsf{View}^{\mathcal{A}}_{\mathcal{C}} \equiv \mathsf{FixView}(\mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathcal{C}^*, \mathsf{Msg}^{\mathcal{A}^*}_{\mathcal{C}^*})$. From the construction of $\mathcal{S}$, we obtain that $\mathsf{View}^{\mathcal{S}}_{\mathcal{C}} \equiv \mathsf{FixView}(\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathcal{C}^*, \mathsf{Msg}^{\mathcal{S}^*}_{\mathcal{C}^*})$. Therefore, we have that the following holds:

$$\begin{aligned}
\mathsf{View}^{\mathcal{S}}_{\mathcal{C}} &\equiv \mathsf{FixView}(\mathsf{View}^{\mathcal{S}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathcal{C}^*, \mathsf{Msg}^{\mathcal{S}^*}_{\mathcal{C}^*}) \\
&\approx \mathsf{FixView}(\mathsf{View}^{\mathcal{A}^*}_{\mathcal{C}^*\cup\mathcal{C}}, \mathcal{C}^*, \mathsf{Msg}^{\mathcal{A}^*}_{\mathcal{C}^*}) \\
&\equiv \mathsf{View}^{\mathcal{A}}_{\mathcal{C}}.
\end{aligned}$$

In the ZK setting, only the verifiers obtain the output. Let $B_{\mathcal{H}}^{\mathcal{S}^*}$ be the outputs of the honest verifiers in the ideal-world execution with $\mathcal{S}^*$, where the honest verifiers either output true or false, or output abort. Similarly, we can also define $B_{\mathcal{H}}^{\mathcal{S}}$. Let $B_{\mathcal{H}}^{\mathcal{A}^*}$ be the outputs of the honest verifiers in the real protocol execution against $\mathcal{A}^*$. As such, we can define $B_{\mathcal{H}}^{\mathcal{A}}$. By the construction of $\mathcal{A}^*$, we obtain that the messages sent by $\mathcal{A}^*$ and $\mathcal{A}$ to the honest verifiers and the (possible) honest prover are the same, and $\mathcal{A}^*$ simulates the verifiers in $\mathcal{C}^*$ honestly. Therefore, together with the definition of FixView, we have

$$\left(\mathsf{View}_{\mathcal{C}}^{\mathcal{A}}, B_{\mathcal{H}}^{\mathcal{A}}\right) \equiv \left(\mathsf{FixView}(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}^{\mathcal{A}^*}, \mathcal{C}^*, \mathsf{Msg}_{\mathcal{C}^*}^{\mathcal{A}^*}), B_{\mathcal{H}}^{\mathcal{A}^*}\right).$$

From $\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mvzk}},\mathcal{S}^*,\mathcal{C}^*\cup\mathcal{C}} \approx \mathsf{REAL}_{\Pi,\mathcal{A}^*,\mathcal{C}^*\cup\mathcal{C}}$, we also have: that the following holds:

$$\left(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}^{\mathcal{S}^*}, B_{\mathcal{H}}^{\mathcal{S}^*}\right) \approx \left(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}^{\mathcal{A}^*}, B_{\mathcal{H}}^{\mathcal{A}^*}\right).$$

Note that $\mathcal{S}$ and $\mathcal{S}^*$ make the same calls to functionality $\mathcal{F}_{\mathsf{mvzk}}$, where the same witness and abort/continue are sent to $\mathcal{F}_{\mathsf{mvzk}}$. Thus, we obtain that $B_{\mathcal{H}}^{\mathcal{S}} \equiv B_{\mathcal{H}}^{\mathcal{S}^*}$. Overall, we have the following:

$$\begin{aligned}\left(\mathsf{View}_{\mathcal{C}}^{\mathcal{S}}, B_{\mathcal{H}}^{\mathcal{S}}\right) &\equiv \left(\mathsf{FixView}(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}^{\mathcal{S}^*}, \mathcal{C}^*, \mathsf{Msg}_{\mathcal{C}^*}^{\mathcal{S}^*}), B_{\mathcal{H}}^{\mathcal{S}^*}\right)\\ &\approx \left(\mathsf{FixView}(\mathsf{View}_{\mathcal{C}^*\cup\mathcal{C}}^{\mathcal{A}^*}, \mathcal{C}^*, \mathsf{Msg}_{\mathcal{C}^*}^{\mathcal{A}^*}), B_{\mathcal{H}}^{\mathcal{A}^*}\right)\\ &\equiv \left(\mathsf{View}_{\mathcal{C}}^{\mathcal{A}}, B_{\mathcal{H}}^{\mathcal{A}}\right).\end{aligned}$$

In conclusion, the real-world execution is indistinguishable from the ideal-world execution, which completes this proof. □

C Information-Theoretic NIMVZK Proof in the Honest-Majority Setting

C.1 Proof of Theorem 1

Theorem 5 (Theorem 1, restated)**.** *Protocol $\Pi_{\mathsf{nimvzk}}^{\mathsf{it}}$ shown in Figure 3 securely realizes functionality $\mathcal{F}_{\mathsf{mvzk}}$ with information-theoretic security and soundness error $\frac{N-1}{|\mathbb{K}|}$ in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{verifyprod}})$-hybrid model in the presence of a malicious adversary corrupting up to a prover and t verifiers.*

Proof. We first consider the case of a malicious prover (i.e., soundness), and then consider the case of an honest prover (i.e., zero knowledge), where exactly t of $n = 2t + 1$ verifiers are assumed to be corrupted in both cases according to Lemma 1. In each case, we construct a simulator $\mathcal{S}$, which is given access to functionality $\mathcal{F}_{\mathsf{mvzk}}$, and runs an adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}_{\mathsf{coin}}$ and $\mathcal{F}_{\mathsf{verifyprod}}$ for $\mathcal{A}$. In the simulation, whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.

Malicious prover. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ and $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with adversary $\mathcal{A}$ as follows:

1. From $i = 1$ to m, for the i-th circuit-input wire, $\mathcal{S}$ receives the shares of $t + 1$ honest verifiers from $\mathcal{A}$, and then reconstructs the *whole* sharing $[w_i]$, which means that the secret w_i as well as the shares of all verifiers (including the shares held by corrupted verifiers) are computed using the $t + 1$ shares of honest verifiers. Then, $\mathcal{S}$ defines a witness $\boldsymbol{w} := (w_1, \ldots, w_m)$.

2. For each addition gate with input sharings $[x]$ and $[y]$, $\mathcal{S}$ computes the whole sharing $[z] := [x] + [y]$.

3. From $i = 1$ to N, for the i-th multiplication gate with whole input sharings $[x_i]$ and $[y_i]$, $\mathcal{S}$ receives the shares of $t + 1$ honest verifiers from $\mathcal{A}$, and then reconstructs the whole output sharing $[z_i]$.

4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ by sending a uniform element $\chi \in \mathbb{K}$ to $\mathcal{A}$, and then computes the whole sharings $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with χ and $\{([x_i], [y_i], [z_i])\}_{i \in [1,N]}$ following the protocol specification.
5. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$ and sends the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ along with $(\boldsymbol{x}, \boldsymbol{y}, z)$ to $\mathcal{A}$. If there exists some $i \in [1, N]$ such that $z_i \neq x_i \cdot y_i$, then $\mathcal{S}$ sends abort to adversary $\mathcal{A}$ and functionality $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts. Otherwise, $\mathcal{S}$ sends accept to $\mathcal{A}$ and continues the simulation.
6. For the single circuit-output gate, $\mathcal{S}$ plays the role of honest verifiers and runs the Open procedure with $\mathcal{A}$. Then, $\mathcal{S}$ sends $\boldsymbol{w}$ to functionality $\mathcal{F}_{\mathsf{mvzk}}$ who returns a decisional result $b \in \{\mathsf{true}, \mathsf{false}\}$ to $\mathcal{S}$. For each honest verifier $\mathcal{V}_i$ simulated by $\mathcal{S}$, if $\mathcal{V}_i$ accepts in the Open procedure, then $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}_{\mathsf{mvzk}}$, otherwise $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is indistinguishable from the ideal-world execution, except with probability $\frac{N-1}{|\mathbb{K}|}$.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ extracts a witness $\boldsymbol{w}$ from $\mathcal{A}$. $\mathcal{S}$ also computes the whole sharing on each wire in the circuit using the shares of $t+1$ honest verifiers.

It is clear that **Hybrid$_1$** has the identical distribution as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ honestly by sending a uniform $\chi \in \mathbb{K}$ to $\mathcal{A}$. Then $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$ as described above. In particular, $\mathcal{S}$ checks whether $z_i = x_i \cdot y_i$ for all $i \in [1, N]$ instead of checking $z = \boldsymbol{x} \odot \boldsymbol{y}$ as done in **Hybrid$_1$**.

It is easy to see that the simulation of $\mathcal{F}_{\mathsf{coin}}$ is perfect. Furthermore, the shares of $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ held by the corrupted verifiers and secrets $(\boldsymbol{x}, \boldsymbol{y}, z)$ that are sent by $\mathcal{F}_{\mathsf{verifyprod}}$ to $\mathcal{A}$ in **Hybrid$_2$** have the identical distribution as that in **Hybrid$_1$**. The only difference between **Hybrid$_1$** and **Hybrid$_2$** is the output of $\mathcal{F}_{\mathsf{verifyprod}}$, where the output depends on whether $z = \boldsymbol{x} \odot \boldsymbol{y}$ in **Hybrid$_1$**, while the output is determined relying on whether $z_i = x_i \cdot y_i$ for all $i \in [1, N]$ in **Hybrid$_2$**. In the following, we prove that checking $z_i = x_i \cdot y_i$ for all $i \in [1, N]$ is equivalent to checking $z = \boldsymbol{x} \odot \boldsymbol{y}$, except with probability $\frac{N-1}{|\mathbb{K}|}$. Consider the following two polynomials of degree-$(N-1)$ over $\mathbb{K}$:

$$F(X) = (x_1 \cdot y_1) + (x_2 \cdot y_2) \cdot X + \cdots + (x_N \cdot y_N) \cdot X^{N-1},$$
$$G(X) = z_1 + z_2 \cdot X + \cdots + z_N \cdot X^{N-1}.$$

Therefore, we have $\boldsymbol{x} \odot \boldsymbol{y} = F(\chi)$ and $z = G(\chi)$ for a uniform element $\chi \in \mathbb{K}$. If there exists some $i \in [1, N]$ such that $z_i \neq x_i \cdot y_i$, then $F(X) - G(X)$ is a non-zero polynomial of degree at most $(N-1)$. According to the Schwartz–Zippel lemma, for a random element $\chi \in \mathbb{K}$, the probability that $\boldsymbol{x} \odot \boldsymbol{y} - z = F(\chi) - G(\chi) = 0$ is at most $\frac{N-1}{|\mathbb{K}|}$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the circuit-output verification phase as described above. In particular, $\mathcal{S}$ sends $\boldsymbol{w}$ to $\mathcal{F}_{\mathsf{mvzk}}$, and determines which honest verifiers obtain the output $b \in \{\mathsf{true}, \mathsf{false}\}$ relying on whether the honest verifier accepts or not during the Open procedure. This is the ideal-world execution.

If an honest verifier $\mathcal{V}_i$ aborts in the Open procedure, $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$ which outputs abort to $\mathcal{V}_i$ in **Hybrid$_3$**. Honest verifier $\mathcal{V}_i$ has the same output (i.e., abort) in **Hybrid$_2$**. If $\mathcal{V}_i$ does not abort, then it will obtain an output bit $\eta \in \{\mathsf{true}, \mathsf{false}\}$ computed following the protocol specification in **Hybrid$_2$**,[1] while

[1]If $\eta = 0$, define $\eta = \mathsf{true}$. If $\eta = 1$, set $\eta = \mathsf{false}$.

it will receive $b \in \{\mathsf{true}, \mathsf{false}\}$ from functionality $\mathcal{F}_{\mathsf{mvzk}}$ in **Hybrid**$_3$. Therefore, we only need to bound the probability that $b \equiv \eta$. If honest verifier $\mathcal{V}_i$ does not abort in **Hybrid**$_2$, then all multiplication gates are computed correctly, and the output bit is also opened correctly. The evaluation of addition gates is trivially correct. Therefore, $\mathcal{V}_i$ will always obtain the correct output bit $\eta = C(\boldsymbol{w})$ in **Hybrid**$_2$, which has the same distribution as b in **Hybrid**$_3$.

Honest prover. If $\mathcal{S}$ receives false from functionality $\mathcal{F}_{\mathsf{mvzk}}$, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ and $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with $\mathcal{A}$ as follows:

1. For each circuit-input wire on an unknown value $w \in \mathbb{F}$, $\mathcal{S}$ samples t uniform elements in $\mathbb{F}$ as the shares of $[w]$ held by corrupted verifiers, and then sends them to $\mathcal{A}$.
2. From $i = 1$ to N, for the i-th multiplication gate with input sharings $[x_i], [y_i]$, $\mathcal{S}$ samples t uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on the output sharing $[z_i]$, and then sends them to $\mathcal{A}$.
3. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ by sending a uniform element $\chi \in \mathbb{K}$ to $\mathcal{A}$, and computes the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ with χ and the shares of corrupted verifiers on $\{([x_i], [y_i], [z_i])\}_{i \in [1,N]}$ following the protocol specification.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$ by sending the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to $\mathcal{A}$ and always outputting accept to $\mathcal{A}$.
5. $\mathcal{S}$ uses the shares of t corrupted verifiers on the sharing $[\eta]$ of the single circuit-output gate along with the secret $\eta = 0$ to compute the shares of $[\eta]$ held by honest verifiers.
6. $\mathcal{S}$ uses the shares of honest verifiers on $[\eta]$ to run the Open procedure with $\mathcal{A}$. For each $i \in \mathcal{H}$, if $\mathcal{V}_i$ accepts in the Open procedure, then $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}_{\mathsf{mvzk}}$, otherwise $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution has the identical distribution as the ideal-world execution, meaning that protocol $\Pi^{\mathsf{it}}_{\mathsf{nimvzk}}$ is perfect zero-knowledge.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ emulates two functionalities $\mathcal{F}_{\mathsf{coin}}$ and $\mathcal{F}_{\mathsf{verifyprod}}$ as described above. Particularly, $\mathcal{S}$ sends the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to $\mathcal{A}$, and also sends accept to $\mathcal{A}$.

Clearly, the simulation of $\mathcal{F}_{\mathsf{coin}}$ is the same in both **Hybrid**$_0$ and **Hybrid**$_1$. Besides, the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ are computed following the protocol description in **Hybrid**$_1$, and thus have the identical distribution as that in **Hybrid**$_0$. In **Hybrid**$_0$, $\mathcal{F}_{\mathsf{verifyprod}}$ checks whether $z = \boldsymbol{x} \odot \boldsymbol{y}$ and returns the result to $\mathcal{A}$, while in **Hybrid**$_1$, $\mathcal{F}_{\mathsf{verifyprod}}$ always returns accept to $\mathcal{A}$. In **Hybrid**$_0$, the output sharings $\{[z_i]\}_{i \in [1,N]}$ of all multiplication gates are computed correctly by the honest prover. Thus, we always have that $z_i = x_i \cdot y_i$ for all $i \in [1, N]$. Following the definition of inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$, we obtain that $z = \boldsymbol{x} \odot \boldsymbol{y}$. Therefore, $\mathcal{F}_{\mathsf{verifyprod}}$ will always return accept to $\mathcal{A}$ in **Hybrid**$_0$. This means that **Hybrid**$_1$ has the same distribution as **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the circuit-output verification phase as described above. In particular, $\mathcal{S}$ uses 0 along with the shares of t corrupted verifiers to reconstruct the whole sharing $[\eta]$ on the circuit-output wire. Then, $\mathcal{S}$ runs the Open procedure with $\mathcal{A}$ using the shares of honest verifiers on $[\eta]$. For each $i \in \mathcal{H}$, $\mathcal{S}$ sends $\mathsf{continue}_i$ or abort_i to $\mathcal{F}_{\mathsf{mvzk}}$ depending on whether $\mathcal{V}_i$ accepts or not in the Open procedure.

In **Hybrid**$_1$, all multiplication gates are computed correctly, and thus the verifiers corrupted by $\mathcal{A}$ obtain an output bit $C(\boldsymbol{w}) = 0$. In **Hybrid**$_2$, $\mathcal{S}$ reconstructs the whole sharing $[\eta]$ such that $\eta = 0$. Therefore, the

Functionality $\mathcal{F}_{\mathsf{inner\text{-}prod}}$

Let $[\boldsymbol{x}]$, $[\boldsymbol{y}]$ be the input vectors of sharings over a field $\mathbb{K}$. Let $\mathcal{C}$ be the set of corrupted verifiers. This functionality runs with a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$, and operates as follows:

1. Upon receiving the shares of $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ from honest verifiers, execute the following:
   - Reconstruct the secrets $\boldsymbol{x}, \boldsymbol{y}$ from the shares.
   - Compute the shares of $[\boldsymbol{x}]$, $[\boldsymbol{y}]$ held by corrupted verifiers, and send these shares to the adversary.
   - If $\mathcal{P}$ is corrupted, send $(\boldsymbol{x}, \boldsymbol{y})$ to the adversary.
2. Receive an additive error $d \in \mathbb{K}$ from the adversary. Also receive a set of shares $\{z^i\}_{i\in\mathcal{C}}$ for corrupted verifiers from the adversary.
3. Compute $z := \boldsymbol{x} \odot \boldsymbol{y} + d \in \mathbb{K}$, and then run $[z] \leftarrow \mathsf{Share}(z)$ such that the share of corrupted verifier $\mathcal{V}_i$ is z^i for $i \in \mathcal{C}$. Then, distribute the shares of $[z]$ to honest verifiers.

Figure 12: **Inner-product functionality with additive errors.**

simulation of the Open procedure in $\textbf{Hybrid}_2$ has the identical distribution as that in $\textbf{Hybrid}_1$. Note that the outputs of honest verifiers in $\textbf{Hybrid}_2$ have the identical distribution as that in $\textbf{Hybrid}_1$. Thus, $\textbf{Hybrid}_2$ is perfectly indistinguishable from $\textbf{Hybrid}_1$.

$\textbf{Hybrid}_3$**:** In this hybrid, $\mathcal{S}$ acts as the honest prover, samples uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on the output sharings of all circuit-input gates and multiplication gates, and send them to $\mathcal{A}$, where the shares held by corrupted verifiers on the output sharings of addition gates are computed locally. This is the ideal-world execution.

In $\textbf{Hybrid}_2$, $\mathcal{S}$ uses the real witness and actual wire values to generate the output sharings of all circuit-input gates and multiplication gates, while in $\textbf{Hybrid}_3$, $\mathcal{S}$ simply samples random elements as the shares of corrupted verifiers. According to the security of the underlying threshold secret sharing scheme, the distribution of the shares of corrupted verifiers in $\textbf{Hybrid}_3$ is the same as that in $\textbf{Hybrid}_2$. Therefore, $\textbf{Hybrid}_3$ has the identical distribution as $\textbf{Hybrid}_2$.

In conclusion, the real-world execution is indistinguishable from the ideal-world execution except with probability $\frac{N-1}{|\mathbb{K}|}$, which completes the proof. □

C.2 Information-Theoretic Verification of Inner-Product Tuples

In this subsection, we describe the information-theoretic protocol to verify the correctness of inner-product tuples. In particular, the prover provides the random sharings to help n verifiers computing the inner-product sharings and a random multiplication triple. For the dimension N of vectors, we can achieve the communication complexity $O((n + \tau) \log_\tau N)$ and the round complexity $O(\log_\tau N)$ for some parameter $\tau \in \mathbb{N}$ (e.g., $\tau = 64$). First of all, we present two useful sub-protocols to compute the inner product of two vectors and compress the dimension of inner-product vectors, respectively.

C.2.1 Prover-Aided Inner-Product Protocol

In the prover-aided setting, we show an information-theoretic inner-product protocol, which securely realizes the functionality $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ shown in Figure 12. In this functionality, we allow the adversary to add an additive error into the output, where additive errors will be detected in our subsequent zero-knowledge

Protocol $\Pi^{\mathsf{it}}_{\mathsf{inner\text{-}prod}}$

Inputs: Every verifier $\mathcal{V}_i$ holds the shares of $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ over a field $\mathbb{K}$ for $i \in [1, n]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}_1, \ldots, \mathcal{V}_n$ execute the following:

1. $\mathcal{P}$ samples $r \leftarrow \mathbb{K}$, and runs $[r] \leftarrow \mathsf{Share}_t(r)$ and $\langle r \rangle \leftarrow \mathsf{Share}_{2t}(r)$ which distribute the shares to all verifiers.
2. All verifiers locally compute $\langle u \rangle := [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r \rangle$.
3. For $i \neq 1$, every verifier $\mathcal{V}_i$ sends its share of $\langle u \rangle$ to $\mathcal{V}_1$, who reconstructs the secret $u = \boldsymbol{x} \odot \boldsymbol{y} + r$.
4. $\mathcal{V}_1$ runs $[u] \leftarrow \mathsf{Share}_t(u)$ that distributes the shares to all verifiers, where $\mathcal{V}_1$ sets the shares of a predetermined set of t verifiers (excluding $\mathcal{V}_1$) as 0.
5. All verifiers locally compute $[z] := [u] - [r]$ and output $[z]$.

Figure 13: **Information-theoretic prover-aided inner-product protocol secure up to additive errors.**

verification protocol. Similar to $\mathcal{F}_{\mathsf{mvzk}}$, if the prover is corrupted, $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ reveals the secret vectors to the adversary.

In Figure 13, we describe the prover-aided inner-product protocol $\Pi^{\mathsf{it}}_{\mathsf{inner\text{-}prod}}$ with information-theoretic security. This protocol builds on the inner-product variant of the original DN multiplication protocol [DN07], which has been used in several previous MPC works such as [CGH+18, GSZ20]. The main difference of our protocol is that the prover helps n verifiers to generate a sharing on the inner-product of two vectors, by providing the *random double sharings* [DN07], i.e., $([r], \langle r \rangle)$ for a random $r \in \mathbb{F}$. Here $[r]$ denotes a degree-t sharing, and $\langle r \rangle$ to denote a degree-$2t$ sharing. By letting the prover aid to generate random double sharings, protocol $\Pi^{\mathsf{it}}_{\mathsf{inner\text{-}product}}$ reduces the communication by at least 0.5 elements per inner product, compared to the state-of-the-art inner-product protocol [GLO+21] in the MPC setting that requires the amortized communication of 4 elements for n inner products. In the protocol $\Pi^{\mathsf{it}}_{\mathsf{inner\text{-}product}}$ shown in Figure 13, $[\boldsymbol{x}] \odot [\boldsymbol{y}]$ denotes that every verifier locally computes the inner product of its shares on $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$, and the resulting shares constitute a degree-$2t$ sharing $\langle \boldsymbol{x} \odot \boldsymbol{y} \rangle$.

Theorem 6. *Protocol $\Pi^{\mathsf{it}}_{\mathsf{inner\text{-}prod}}$ shown in Figure 13 securely realizes functionality $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ in the presence of a malicious adversary corrupting up to a prover and exactly t verifiers.*

Proof. For any adversary $\mathcal{A}$, we construct a simulator $\mathcal{S}$, which is given access to $\mathcal{F}_{\mathsf{inner\text{-}prod}}$, and runs $\mathcal{A}$ as a subroutine. In the simulation, whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{inner\text{-}prod}}$, and also aborts.

Description of simulation. Given access to functionality $\mathcal{F}_{\mathsf{inner\text{-}prod}}$, $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1. $\mathcal{S}$ simulates the generation of random double sharings as follows:
   - If $\mathcal{P}$ is honest, $\mathcal{S}$ samples $2t + 1$ uniform elements in $\mathbb{K}$ as the shares of all verifiers on a degree-$2t$ sharing $\langle r \rangle$, and reconstructs the secret r. Then, $\mathcal{S}$ uses the same secret r to generate a random degree-t sharing $[r]$ by running $\mathsf{Share}_t(r)$, and sends the shares of $([r], \langle r \rangle)$ to the corrupted verifiers.
   - If $\mathcal{P}$ is corrupted, $\mathcal{S}$ receives the shares of honest verifiers on $([r], \langle r \rangle)$ from $\mathcal{A}$, and then reconstructs the whole sharing $[r]$ using the shares of $t + 1$ honest verifiers.
2. $\mathcal{S}$ receives the shares of corrupted verifiers on $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ from $\mathcal{F}_{\mathsf{inner\text{-}prod}}$.
3. $\mathcal{S}$ computes the shares of honest verifiers on $\langle u \rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r \rangle$ and the secret u as follows:

- If $\mathcal{P}$ is honest, then for every honest verifier, $\mathcal{S}$ samples a uniform element in $\mathbb{K}$ as its share on $\langle u\rangle$. Moreover, $\mathcal{S}$ computes the shares of corrupted verifiers on $\langle u\rangle$ by using their shares on $[\boldsymbol{x}]$, $[\boldsymbol{y}]$ and $\langle r\rangle$. Then, $\mathcal{S}$ uses the shares of all verifiers on $\langle u\rangle$ to reconstruct u.
- If $\mathcal{P}$ is corrupted, $\mathcal{S}$ receives the secret vectors $\boldsymbol{x}, \boldsymbol{y}$ from $\mathcal{F}_{\textsf{inner-prod}}$, and reconstructs the whole sharings $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ using vectors $\boldsymbol{x}, \boldsymbol{y}$ and the shares of t corrupted verifiers. In this case, $\mathcal{S}$ computes the shares of honest verifiers on $\langle u\rangle$ following the protocol specification, and also computes $u = \boldsymbol{x} \odot \boldsymbol{y} + r$.

4. $\mathcal{S}$ computes an additive error d and the shares of corrupted verifiers on $[z] = [\boldsymbol{x} \odot \boldsymbol{y} + d]$ as follows:

   - If $\mathcal{V}_1$ is honest, then $\mathcal{S}$ receives the shares of corrupted verifiers on $\langle u\rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r\rangle$. Then, $\mathcal{S}$ reconstructs the secret u' using these shares and the shares of honest verifiers on $\langle u\rangle$. Next, $\mathcal{S}$ generates the sharing $[u']$ following the protocol description, and sends the shares to the corrupted verifiers.
   - If $\mathcal{V}_1$ is corrupted, then $\mathcal{S}$ sends the shares of honest verifiers on $\langle u\rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r\rangle$ to $\mathcal{A}$. Then, $\mathcal{S}$ receives the shares of honest verifiers on $[u]$ from $\mathcal{A}$ under the constraint that the share of an honest verifier is 0 if it is in a predetermined set of t verifiers. Next, $\mathcal{S}$ uses these shares to reconstruct the whole sharing $[u']$ (involving the secret u').

   $\mathcal{S}$ computes the additive error $d := u' - u$, and also computes the shares of corrupted verifiers on $[z] = [u'] - [r]$ using their shares on $[u']$ and $[r]$.

5. $\mathcal{S}$ sends the additive error d and the shares of $[z]$ held by corrupted verifiers to functionality $\mathcal{F}_{\textsf{inner-prod}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is perfectly indistinguishable from the ideal-world execution.

$\textbf{Hybrid}_0$**:** This is the real-world execution.

$\textbf{Hybrid}_1$**:** In this hybrid, $\mathcal{S}$ simulates the generation of random double sharings as described above. In particular, if $\mathcal{P}$ is honest, $\mathcal{S}$ samples $2t + 1$ uniform elements to reconstruct a degree-$2t$ whole sharing $\langle r\rangle$, and then uses the same secret r to generate a random degree-t sharing $[r]$. Besides, $\mathcal{S}$ receives the shares of corrupted verifiers on $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ from $\mathcal{F}_{\textsf{inner-prod}}$.

The only difference between $\textbf{Hybrid}_0$ and $\textbf{Hybrid}_1$ is that the generation manner of random double sharings in the case that $\mathcal{P}$ is honest. Specifically, the honest prover $\mathcal{P}$ samples $r \leftarrow \mathbb{K}$ and generates a pair of random double sharings $([r], \langle r\rangle)$ by running the Share procedure in $\textbf{Hybrid}_0$, while $\mathcal{S}$ samples $2t+1$ random shares to reconstruct $\langle r\rangle$ and uses the same secret r to generate $[r]$ in $\textbf{Hybrid}_1$. For a random element r, the two manners of generating $([r], \langle r\rangle)$ are the same. Therefore, $\textbf{Hybrid}_1$ has the identical distribution as $\textbf{Hybrid}_0$.

$\textbf{Hybrid}_2$**:** In this hybrid, $\mathcal{S}$ computes an additive error d and the shares of corrupted verifiers on $[z] = [\boldsymbol{x} \odot \boldsymbol{y} + d]$ as described above. Then, $\mathcal{S}$ sends d and these shares of $[z]$ to $\mathcal{F}_{\textsf{inner-prod}}$.

The only difference between $\textbf{Hybrid}_1$ and $\textbf{Hybrid}_2$ is that in $\textbf{Hybrid}_1$, the shares of honest verifiers on $[z]$ are computed following the protocol description, while in $\textbf{Hybrid}_2$, the honest verifiers obtain their shares on $[z]$ from $\mathcal{F}_{\textsf{inner-prod}}$. Note that the shares of $[z]$ held by honest verifiers are determined by the shares of corrupted verifiers and the secret $\boldsymbol{x} \odot \boldsymbol{y} + d$. Therefore, the distribution of $\textbf{Hybrid}_2$ is the same as $\textbf{Hybrid}_1$.

$\textbf{Hybrid}_3$**:** In this hybrid, $\mathcal{S}$ computes the shares of honest verifiers on $\langle u\rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r\rangle$ and the secret u as described above. In particular, if $\mathcal{P}$ is honest, then $\mathcal{S}$ samples uniform elements as the shares of honest verifiers on $\langle u\rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r\rangle$. This is the ideal-world execution.

The only difference between $\textbf{Hybrid}_2$ and $\textbf{Hybrid}_3$ is that if $\mathcal{P}$ is honest, $\mathcal{S}$ uses the real shares of $\langle u\rangle$ held by honest verifiers in $\textbf{Hybrid}_2$, while $\mathcal{S}$ samples uniform elements as the shares of honest verifiers on $\langle u\rangle$ in

Protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$

Inputs: $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold the shares of $([\boldsymbol{x}_1], [\boldsymbol{y}_1], [z_1]), \ldots, ([\boldsymbol{x}_m], [\boldsymbol{y}_m], [z_m])$ over a field $\mathbb{K}$.

Protocol execution: Prover $\mathcal{P}$ and all verifiers execute as follows:

1. All verifiers locally compute $[\boldsymbol{f}(\cdot)]$ and $[\boldsymbol{g}(\cdot)]$ with $[\boldsymbol{x}_i]$ and $[\boldsymbol{y}_i]$ for all $i \in [1, m]$ respectively, where $\boldsymbol{f}(\cdot)$ and $\boldsymbol{g}(\cdot)$ are vectors of degree-$(m-1)$ polynomials such that $\boldsymbol{f}(i) = \boldsymbol{x}_i$ and $\boldsymbol{g}(i) = \boldsymbol{y}_i$ for $i \in [1, m]$. For each $i \in [m+1, 2m-1]$, all verifiers also locally compute $[\boldsymbol{f}(i)]$ and $[\boldsymbol{g}(i)]$.
2. For $i \in [m+1, 2m-1]$, $\mathcal{P}$ and all verifiers call $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ on $([\boldsymbol{f}(i)], [\boldsymbol{g}(i)])$ to compute $[z_i] = [\boldsymbol{f}(i) \odot \boldsymbol{g}(i)]$.
3. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomial such that $h(i) = z_i$ for $i \in [1, 2m-1]$. All verifiers locally compute $[h(\cdot)]$ by using sharings $[z_1], \ldots, [z_{2m-1}]$.
4. All verifiers call $\mathcal{F}_{\mathsf{coin}}$ to sample a random element $\alpha \in \mathbb{K}$. If $\alpha \in [1, m]$, the verifiers abort. Otherwise, the verifiers output $([\boldsymbol{f}(\alpha)], [\boldsymbol{g}(\alpha)], [h(\alpha)])$.

Figure 14: **Protocol for compressing inner-product tuples in the $(\mathcal{F}_{\mathsf{inner\text{-}prod}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model.**

Hybrid$_3$. In the case that $\mathcal{P}$ is honest, the shares of $\langle r \rangle$ held by honest verifiers are uniformly random, and thus the shares of honest verifiers on $\langle u \rangle = [\boldsymbol{x}] \odot [\boldsymbol{y}] + \langle r \rangle$ are also random. Therefore, the distribution of **Hybrid$_3$** is identical to that of **Hybrid$_2$**.

Overall, the real-world execution has the identical distribution as the ideal-world execution, which completes the proof. □

C.2.2 Prover-Aided Verification for Inner-Product Tuples

Firstly, we show an efficient sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ shown in Figure 14 in the $(\mathcal{F}_{\mathsf{inner\text{-}prod}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model, which compresses m inner-product tuples into a single inner-product tuple. This protocol is based on the inner-product extension of the batch-wise multiplication verification technique [BFO12], which has been used in honest-majority MPC protocols [NV18, GSZ20]. The main difference for $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ is that it works in the MVZK setting and can adopt the prover-aided inner-product protocol described in the previous subsection to instantiate $\mathcal{F}_{\mathsf{inner\text{-}prod}}$.

Lemma 3. *If there exists at least one incorrect inner-product tuple and protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ does not abort, then the resulting tuple output by $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ is correct with probability at most $\frac{2(m-1)}{|\mathbb{K}|-m}$.*

Proof. If $\alpha \in [1, m]$, then $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ aborts. Thus, there are $|\mathbb{K}| - m$ choices of α that do not cause an abort of $\Pi^{\mathsf{it}}_{\mathsf{compress}}$. Since there are at least one incorrect inner-product tuple, we have that $\boldsymbol{f} \odot \boldsymbol{g} \neq h$. Note that the polynomial $\boldsymbol{f} \odot \boldsymbol{g} - h$ has a degree at most $2(m-1)$. Under the condition that $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ does not abort, according to the Schwartz–Zippel lemma, for a random element $\alpha \in \mathbb{K}$, the probability that $\boldsymbol{f}(\alpha) \odot \boldsymbol{g}(\alpha) - h(\alpha) = 0$ is bounded by $\frac{2(m-1)}{|\mathbb{K}|-m}$. That is, if $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ does not abort, it outputs a correct inner-product tuple with probability at most $\frac{2(m-1)}{|\mathbb{K}|-m}$, which completes the proof. □

Below, we describe the information-theoretic verification protocol for inner-product tuples in the prover-aided setting. This protocol invokes functionality $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ and sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$, and is shown in Figure 15, where the prover samples random sharings to aid the verification without affecting the soundness, while keeping the secrets zero-knowledge. This protocol is divided into two phases: 1) the first phase is to *recursively* reduce the dimension of the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$; 2) the second phase is to randomize the inner-product tuple with a small dimension that is the final output in the first phase, and then

Protocol $\Pi^{\mathsf{it}}_{\mathsf{verifyprod}}$

Inputs: $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold an inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ defined over a field $\mathbb{K}$, where the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is N. Let τ be the compression parameter.

- **Dimension-reduction:** $\mathcal{P}$ and all verifiers initialize the dimension $m := N$ of the inner-product tuple, and iteratively execute the following steps till $m \leq \tau$.
  1. All verifiers parse $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$ as $[\boldsymbol{x}] = ([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\tau])$ and $[\boldsymbol{y}] = ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\tau])$ respectively, where $\boldsymbol{a}_i, \boldsymbol{b}_i$ are vectors of dimension $\ell = m/\tau$ for $i \in [1, \tau]$.
  2. For $i \in [1, \tau - 1]$, $\mathcal{P}$ and all verifiers call $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ on $([\boldsymbol{a}_i], [\boldsymbol{b}_i])$ to compute $[c_i] = [\boldsymbol{a}_i \odot \boldsymbol{b}_i]$. Then, the verifiers set $[c_\tau] := [z] - \sum_{i \in [1, \tau-1]} [c_i]$.
  3. $\mathcal{P}$ and all verifiers execute the sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ shown in Figure 14 on $\{([\boldsymbol{a}_i], [\boldsymbol{b}_i], [c_i])\}_{i \in [1, \tau]}$.
  4. The verifiers update $([\boldsymbol{x}], [\boldsymbol{y}], [z]) := ([\boldsymbol{a}], [\boldsymbol{b}], [c])$, which is the output from $\Pi^{\mathsf{it}}_{\mathsf{compress}}$, and set $m := m/\tau$.
- **Randomization:** Let $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ be the inner-product tuple output in the previous phase, where the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is $m \leq \tau$. Prover $\mathcal{P}$ and all verifiers execute the following:
  1. All verifiers parse $[\boldsymbol{x}] = ([a_1], \ldots, [a_m])$ and $[\boldsymbol{y}] = ([b_1], \ldots, [b_m])$.
  2. $\mathcal{P}$ samples $a_0, b_0 \leftarrow \mathbb{K}$ and computes $c_0 := a_0 \cdot b_0$. Then, $\mathcal{P}$ runs $[v] \leftarrow \mathsf{Share}(v)$ for each $v \in \{a_0, b_0, c_0\}$, which distributes the shares to all verifiers.
  3. For $i \in [1, m - 1]$, $\mathcal{P}$ and all verifiers call $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ on $([a_i], [b_i])$ to compute $[c_i] = [a_i \cdot b_i]$. Then, the verifiers set $[c_m] := [z] - \sum_{i \in [1, m-1]} [c_i]$.
  4. $\mathcal{P}$ and all verifiers execute the sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ shown in Figure 14 on $\{([a_i], [b_i], [c_i])\}_{i \in [0, m]}$. Then, all verifiers obtain the output $([a], [b], [c])$.
  5. All verifiers run $v \leftarrow \mathsf{Open}([v])$ for each $v \in \{a, b, c\}$. If abort is received in the Open procedure, the verifiers abort. Then, the verifiers check that $c = a \cdot b$. If the check fails, the verifiers output abort. Otherwise, they output accept.

Figure 15: **Information-theoretic verification protocol for inner-product tuples in the prover-aided setting.**

to check correctness of the tuple by the Open procedure. While the randomization technique has been used in prior work [NV18, BBC$^+$19, GSZ20], we use it in the prover-aided setting, where the prover generates a random multiplication triple. In the dimension-reduction phase, for each iteration, if the dimension m is *not* a multiple of τ, then $\tau \cdot \lceil m/\tau \rceil - m$ zero sharings $[0]$ are generated *locally* by all verifiers and added into the vectors of sharings $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$.

Theorem 7. *Protocol $\Pi^{\mathsf{it}}_{\mathsf{verifyprod}}$ shown in Figure 15 securely realizes functionality $\mathcal{F}_{\mathsf{verifyprod}}$ with information-theoretic security and soundness error $\frac{2\tau \lceil \log_\tau N \rceil}{|\mathbb{K}| - \tau}$ in the $(\mathcal{F}_{\mathsf{inner\text{-}prod}}, \mathcal{F}_{\mathsf{coin}})$-hybrid model in the presence of a malicious adversary corrupting up to the prover and exactly t verifiers.*

Proof. For any adversary $\mathcal{A}$, we construct a simulator $\mathcal{S}$, which is given access to functionality $\mathcal{F}_{\mathsf{verifyprod}}$ and runs $\mathcal{A}$ as a subroutine. Whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{verifyprod}}$ and also aborts.

Description of simulation. $\mathcal{S}$ receives the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ from functionality $\mathcal{F}_{\mathsf{verifyprod}}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also receives from $\mathcal{F}_{\mathsf{verifyprod}}$ the secrets $\boldsymbol{x}, \boldsymbol{y}, z$, and computes $d := z - \boldsymbol{x} \odot \boldsymbol{y}$. Otherwise, $\mathcal{S}$ sets $d = 0$. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ and $\mathcal{F}_{\mathsf{inner\text{-}prod}}$, and interacts with $\mathcal{A}$ as follows:

- *Simulation of sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$:* Given $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [z_i])\}_{i \in [1, m]}$ as input, $\mathcal{S}$ simulates a protocol execution of $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ as follows:

1. $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{f}(\cdot)]$ and $[\boldsymbol{g}(\cdot)]$ using their shares on $\{([\boldsymbol{x}_i],[\boldsymbol{y}_i])\}_{i\in[1,m]}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ computes the polynomials $\boldsymbol{f}(\cdot)$ and $\boldsymbol{g}(\cdot)$.
2. For $i \in [m+1, 2m-1]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ and sends the shares of $[\boldsymbol{f}(i)]$ and $[\boldsymbol{g}(i)]$ held by corrupted verifiers to $\mathcal{A}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ sends $\boldsymbol{f}(i)$ and $\boldsymbol{g}(i)$ for all $i \in [m+1, 2m-1]$ to $\mathcal{A}$. Then, $\mathcal{S}$ receives the shares of corrupted verifiers on $[z_i] = [\boldsymbol{f}(i) \odot \boldsymbol{g}(i)]$ along with an additive error d_i from $\mathcal{A}$.
3. $\mathcal{S}$ computes the shares of corrupted verifiers on $[h(\cdot)]$ using their shares on $\{[z_i]\}_{i\in[1,2m-1]}$.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{coin}}$ by sending a uniform element α to $\mathcal{A}$. If $\alpha \in [1,m]$, $\mathcal{S}$ sends abort to functionality $\mathcal{F}_{\mathsf{verifyprod}}$ and then aborts. Otherwise, $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{a}] = [\boldsymbol{f}(\alpha)]$, $[\boldsymbol{b}] = [\boldsymbol{g}(\alpha)]$ and $[c] = [h(\alpha)]$ using their shares on $[\boldsymbol{f}(\cdot)]$, $[\boldsymbol{g}(\cdot)]$ and $[h(\cdot)]$. Besides, $\mathcal{S}$ uses the errors $\{d_i\}_{i\in[1,2m-1]}$ to reconstruct an additive error d' with $d' = c - \boldsymbol{a} \odot \boldsymbol{b}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ computes $\boldsymbol{a} = \boldsymbol{f}(\alpha)$, $\boldsymbol{b} = \boldsymbol{g}(\alpha)$ and $c = h(\alpha) = \boldsymbol{a} \odot \boldsymbol{b} + d'$.

- *Simulation of dimension-reduction*: Following the protocol specification, $\mathcal{S}$ simulates the view of $\mathcal{A}$ in the following iterative manner:

  1. For $i \in [1,\tau]$, $\mathcal{S}$ computes the shares of $[\boldsymbol{a}_i]$, $[\boldsymbol{b}_i]$ held by corrupted verifiers directly from the shares of corrupted verifiers on $[\boldsymbol{x}]$, $[\boldsymbol{y}]$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ uses $\boldsymbol{x}, \boldsymbol{y}$ to compute $\boldsymbol{a}_i, \boldsymbol{b}_i$ for $i \in [1,\tau]$.
  2. For $i \in [1,\tau-1]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ and sends the shares of corrupted verifiers on $[\boldsymbol{a}_i]$, $[\boldsymbol{b}_i]$ to $\mathcal{A}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also sends $\boldsymbol{a}_i, \boldsymbol{b}_i$ to $\mathcal{A}$. Then, for $i \in [1,\tau-1]$, $\mathcal{S}$ receives the shares of $[c_i]$ held by corrupted verifiers and an additive error d_i from $\mathcal{A}$.
  3. $\mathcal{S}$ computes the shares of corrupted verifiers on $[c_\tau] = [z] - \sum_{i\in[1,\tau-1]}[c_i]$, and also computes $d_\tau := d - \sum_{i\in[1,\tau-1]} d_i$.
  4. $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ on the input inner-product tuples $\{([\boldsymbol{a}_i],[\boldsymbol{b}_i],[c_i])\}_{i\in[1,\tau]}$ as described above.
  5. $\mathcal{S}$ updates the shares of corrupted verifiers on $([\boldsymbol{x}],[\boldsymbol{y}],[z])$ with their shares of $([\boldsymbol{a}],[\boldsymbol{b}],[c])$, and also updates $d := d'$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ updates $(\boldsymbol{x},\boldsymbol{y},z)$ as $(\boldsymbol{a},\boldsymbol{b},c)$. Then, $\mathcal{S}$ performs the next iteration.

- *Simulation of randomization:* $\mathcal{S}$ knows the shares of corrupted verifiers on $([\boldsymbol{x}],[\boldsymbol{y}],[z])$ output in the dimension-reduction phase, where the dimension of vectors $\boldsymbol{x}, \boldsymbol{y}$ is $m \le \tau$. Furthermore, $\mathcal{S}$ knows the additive error $d = z - \boldsymbol{x} \odot \boldsymbol{y}$, and gets $(\boldsymbol{x},\boldsymbol{y},z)$ if $\mathcal{P}$ is corrupted.

  1. $\mathcal{S}$ computes the shares of corrupted verifiers on $[a_i]$, $[b_i]$ for $i \in [1,m]$ from their shares of $[\boldsymbol{x}]$, $[\boldsymbol{y}]$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also computes a_i, b_i for all $i \in [1,m]$.
  2. If $\mathcal{P}$ is honest, $\mathcal{S}$ samples uniform elements as the shares of $([a_0],[b_0],[c_0])$ held by corrupted verifiers, and then sends them to $\mathcal{A}$. Otherwise, $\mathcal{S}$ receives from $\mathcal{A}$ the shares of $t+1$ honest verifiers on $([a_0],[b_0],[c_0])$, and then reconstructs the *whole* sharings $[a_0],[b_0],[c_0]$. In both cases, $\mathcal{S}$ computes $d_0 := c_0 - a_0 \cdot b_0$, where $d_0 = 0$ for the case of honest $\mathcal{P}$.
  3. For $i \in [1,m-1]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ and sends the shares of corrupted verifiers on $[a_i]$, $[b_i]$ to $\mathcal{A}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also sends a_i, b_i to $\mathcal{A}$. Then, for $i \in [1,m-1]$, $\mathcal{S}$ receives the shares of $[c_i]$ held by corrupted verifiers and an additive error d_i from $\mathcal{A}$.
  4. $\mathcal{S}$ computes the shares of corrupted verifiers on $[c_m] = [z] - \sum_{i\in[1,m-1]}[c_i]$, and sets $d_m := d - \sum_{i\in[1,m-1]} d_i$.
  5. $\mathcal{S}$ simulates the execution of sub-protocol $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ on $\{([a_i],[b_i],[c_i])\}_{i\in[0,m]}$ as described above. In particular, $\mathcal{S}$ obtains the shares of corrupted verifiers on the output sharings $([a],[b],[c])$, and also gets the additive error $d' = c - a \cdot b$ depending on the errors $d_0, d_1, \ldots, d_m$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ also knows the secrets a, b, c.

6. If $\mathcal{P}$ is honest, $\mathcal{S}$ samples a, b uniformly at random and computes $c = a \cdot b + d'$. Otherwise, $\mathcal{S}$ has known a, b, c. Then, $\mathcal{S}$ uses the secrets a, b, c along with the shares of corrupted verifiers to reconstruct the whole sharings $[a], [b], [c]$.
7. $\mathcal{S}$ receives an output either abort or accept from functionality $\mathcal{F}_{\mathsf{verifyprod}}$. $\mathcal{S}$ uses the shares of $[a], [b], [c]$ held by honest verifiers to run the Open procedure with $\mathcal{A}$. Then, $\mathcal{S}$ plays the role of honest verifiers and checks that $c = a \cdot b$. For every honest verifier $\mathcal{V}_i$ (simulated by $\mathcal{S}$), if it aborts, then $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{verifyprod}}$, otherwise $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}_{\mathsf{verifyprod}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is indistinguishable from the ideal-world execution, except with probability $\frac{2\tau\lceil\log_\tau N\rceil}{|\mathbb{K}|-\tau}$.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted verifiers and the additive errors (together with computing the secrets if $\mathcal{P}$ is corrupted) as described above, where $\mathcal{F}_{\mathsf{inner\text{-}prod}}$ and $\mathcal{F}_{\mathsf{coin}}$ are simulated honestly.

The simulation of $\mathcal{S}$ in **Hybrid$_1$** does not change the behaviors of honest parties. Thus, **Hybrid$_1$** has the identical distribution as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, if $\mathcal{P}$ is honest, then $\mathcal{S}$ samples uniform elements as the shares of corrupted verifiers on $([a_0], [b_0], [c_0])$. Furthermore, instead of using the real sharings $[a], [b], [c]$, $\mathcal{S}$ constructs $([a], [b], [c])$ with uniform elements a, b as described above. Note that if $\mathcal{P}$ is corrupted, then $\mathcal{S}$ knows the real secrets a, b, c and thus can reconstruct and use the real sharings $[a], [b], [c]$.

Clearly, the shares of corrupted verifiers on $([a_0], [b_0], [c_0])$ in **Hybrid$_2$** have the same distribution as that in **Hybrid$_1$** for an honest prover $\mathcal{P}$. In the randomization procedure, a and b are linear combinations of $\{a_i\}_{i\in[0,m]}$ and $\{b_i\}_{i\in[0,m]}$ respectively, where the coefficients are all non-zero. Since a_0, b_0 are uniformly random in the case that $\mathcal{P}$ is honest, a, b are also uniformly random. The only difference between **Hybrid$_1$** and **Hybrid$_2$** is that in **Hybrid$_1$**, a and b are masked by random elements a_0 and b_0, while in **Hybrid$_2$**, a, b are randomly sampled by $\mathcal{S}$. Therefore, a, b have the identical distribution in **Hybrid$_1$** and **Hybrid$_2$**. In addition, c is determined by elements a, b and the additive error d', and thus the distribution of c remains the same in two hybrids. Overall, the distribution of **Hybrid$_2$** is identical to that of **Hybrid$_1$**.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the Open procedure and interacts with functionality $\mathcal{F}_{\mathsf{verifyprod}}$ which sends an output or abort to every honest verifier, as described above. This is the ideal-world execution.

The only difference between **Hybrid$_2$** and **Hybrid$_3$** is that if the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect for a malicious prover, an honest verifier outputs accept in **Hybrid$_2$**, while the honest verifier outputs abort in **Hybrid$_3$**. We bound the difference between two hybrids by the following lemmas.

Lemma 4. *If the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect where the dimension of $\boldsymbol{x}, \boldsymbol{y}$ is N, all honest verifiers will output* abort, *except with probability at most* $\frac{2\tau\lceil\log_\tau N\rceil}{|\mathbb{K}|-\tau}$.

Proof. For the i-th iteration with $i \in [1, \log_\tau N - 1]$ in the dimension-reduction phase, suppose that the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect. We first show that at least one of the following τ inner-product tuples is incorrect:

$$([\boldsymbol{a}_1], [\boldsymbol{b}_1], [c_1]), \ldots, ([\boldsymbol{a}_\tau], [\boldsymbol{b}_\tau], [c_\tau])$$

where $[\boldsymbol{x}] = ([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\tau])$, $[\boldsymbol{y}] = ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\tau])$ and $[z] = \sum_{i\in[1,\tau]}[c_i]$. If at least one of the first $\tau - 1$ inner-product tuples is incorrect, then the statement holds. Otherwise (i.e., all of the first $\tau - 1$ tuples are correct), $\boldsymbol{a}_i \odot \boldsymbol{b}_i = c_i$ for all $i \in [1, \tau - 1]$. Since the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect, we have that $\boldsymbol{x} \odot \boldsymbol{y} \neq z$. Thus, we have the following:

$$\boldsymbol{a}_\tau \odot \boldsymbol{b}_\tau = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i \in [1, \tau-1]} \boldsymbol{a}_i \odot \boldsymbol{b}_i = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i \in [1, \tau-1]} c_i \neq z - \sum_{i \in [1, \tau-1]} c_i = c_\tau.$$

This means that the τ-th inner-product tuple $([\boldsymbol{a}_\tau], [\boldsymbol{b}_\tau], [c_\tau])$ is incorrect. According to Lemma 3, we obtain that the resulting inner-product tuple $([\boldsymbol{a}], [\boldsymbol{b}], [c])$ output by $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ is incorrect except with probability at most $\frac{2(\tau-1)}{|\mathbb{K}|-\tau}$.

In the randomization phase, the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ output by the dimension-reduction phase is incorrect, except with probability at most $\frac{2(\tau-1)(\lceil \log_\tau N \rceil - 1)}{|\mathbb{K}|-\tau}$ from the above analysis, where the dimension of $\boldsymbol{x}, \boldsymbol{y}$ is now $m \leq \tau$. In the following, we assume that the input tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect. Let $([a_1], [b_1], [c_1]), \ldots, ([a_m], [b_m], [c_m])$ be the m multiplication triples such that $[\boldsymbol{x}] = ([a_1], \ldots, [a_m])$, $[\boldsymbol{y}] = ([b_1], \ldots, [b_m])$ and $[z] = \sum_{i \in [1,m]} [c_i]$. Following a similar analysis, we have that at least one of these multiplication triples is incorrect. No matter what correctness of the additional triple $([a_0], [b_0], [c_0])$ is, based on Lemma 3, the resulting multiplication triple $([a], [b], [c])$ output by $\Pi^{\mathsf{it}}_{\mathsf{compress}}$ is incorrect, except with probability at most $\frac{2m}{|\mathbb{K}|-m-1} \leq \frac{2\tau}{|\mathbb{K}|-\tau-1} \approx \frac{2\tau}{|\mathbb{K}|-\tau}$ for a large field $\mathbb{K}$.

Overall, if the input inner-product tuple $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ is incorrect, the resulting multiplication triple $([a], [b], [c])$ output in the randomization phase is also incorrect, except with probability at most

$$\frac{2(\tau-1)(\lceil \log_\tau N \rceil - 1)}{|\mathbb{K}| - \tau} + \frac{2\tau}{|\mathbb{K}| - \tau} \leq \frac{2\tau \lceil \log_\tau N \rceil}{|\mathbb{K}| - \tau}.$$

Note that an incorrect multiplication triple $([a], [b], [c])$ will cause all honest verifiers either abort in the Open procedure, or output abort by checking that $a \cdot b \neq c$. Therefore, all honest verifiers will output abort, except with probability at most $\frac{2\tau \lceil \log_\tau N \rceil}{|\mathbb{K}|-\tau}$, which completes the proof. □

From the above lemma, we have that **Hybrid**$_3$ is indistinguishable from **Hybrid**$_2$, except with probability at most $\frac{2\tau \lceil \log_\tau N \rceil}{|\mathbb{K}|-\tau}$.

By the above hybrids, we obtain that the real-world execution is indistinguishable from the ideal-world execution, except with probability at most $\frac{2\tau \lceil \log_\tau N \rceil}{|\mathbb{K}|-\tau}$, which completes the proof. □

D Proof of Theorem 2

Theorem 8 (Theorem 2, restated)**.** *Let* H_1 *and* H_2 *be two random oracles. Protocol* $\Pi^{\mathsf{fs}}_{\mathsf{snimvzk}}$ *shown in Figure 4 securely realizes functionality* $\mathcal{F}_{\mathsf{mvzk}}$ *with soundness error at most* $\frac{Q_1 n + (Q_2+1)N}{2^\lambda}$ *in the* $\mathcal{F}_{\mathsf{verifyprod}}$*-hybrid model in the presence of a malicious adversary corrupting up to a prover and* t *verifiers, where* Q_1 *and* Q_2 *are the number of queries to random oracles* H_1 *and* H_2 *respectively.*

Proof. According to Lemma 1, we only need to consider that exactly t of $n = 2t+1$ verifiers are corrupted. We first consider the case of a malicious prover (i.e., soundness), and then consider the case of an honest prover (i.e., zero knowledge). In each case, we construct a PPT simulator $\mathcal{S}$, which is given access to functionality $\mathcal{F}_{\mathsf{mvzk}}$, and runs a PPT adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}_{\mathsf{verifyprod}}$ for $\mathcal{A}$. In the simulation, whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.

The simulation of $\mathcal{S}$ for the circuit-evaluation phase and the circuit-output verification phase is the same as that in the proof of Theorem 1. Therefore, we focus on the simulation of $\mathcal{S}$ for the verification of multiplication gates, which is described as follows:

Malicious prover. In the verification phase of multiplication gates, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with $\mathcal{A}$ as follows:

1. After the circuit-evaluation phase, for $i \in [1, m]$, $\mathcal{S}$ gets the whole sharings $[w_i]$. For $i \in [1, N]$, $\mathcal{S}$ also obtains the whole sharings $[x_i], [y_i], [z_i]$ on the i-th multiplication gate.

2. $\mathcal{S}$ simulates the random oracles H_1 and H_2 by responding the queries with random values while keeping the consistency of answers.

3. $\mathcal{S}$ receives from $\mathcal{A}$ the public commitments $com_1, \ldots, com_n$. For $i \in \mathcal{H}$, $\mathcal{S}$ acts as honest verifier $\mathcal{V}_i$ and receives a randomness $r_i \in \{0,1\}^\lambda$ from $\mathcal{A}$.

4. For each $i \in \mathcal{H}$, $\mathcal{S}$ uses the shares of $[w_1], \ldots, [w_m]$ and $[z_1], \ldots, [z_N]$ held by honest verifier $\mathcal{V}_i$ as well as r_i to check the correctness of commitment com_i, where $\mathcal{S}$ makes the corresponding query to H_1 by itself. If the check fails, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.

5. $\mathcal{S}$ sets the public random coefficient $\chi := \mathsf{H}_2(com_1, \ldots, com_n)$ by making the corresponding query to H_2. Then, $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{x}], [\boldsymbol{y}], [z]$, and also computes the corresponding secrets $\boldsymbol{x}, \boldsymbol{y}, z$, following the protocol specification.

6. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$ by sending the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ along with $(\boldsymbol{x}, \boldsymbol{y}, z)$ to $\mathcal{A}$. If there exists some $i \in [1, N]$ such that $z_i \neq x_i \cdot y_i$, then $\mathcal{S}$ sends abort to $\mathcal{A}$ and functionality $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts. Otherwise, $\mathcal{S}$ sends accept to $\mathcal{A}$.

Below, we show that $\mathcal{S}$ perfectly simulates the view of $\mathcal{A}$ in the verification phase of multiplication gates, except with probability at most $\frac{Q_1 n + (Q_2+1)N}{2^\lambda}$. In this phase, the difference between the real-world execution and ideal-world execution is that the real-world execution checks that $z = \boldsymbol{x} \odot \boldsymbol{y}$ when emulating $\mathcal{F}_{\mathsf{verifyprod}}$, while the ideal-world execution checks that $z_i = x_i \cdot y_i$ for $i \in [1, N]$. Let E_1 be the event that for some $i \in \mathcal{H}$, $com_i = \mathsf{H}_1(\mathsf{Msg}_i, r_i)$ and (Msg_i, r_i) was queried to H_1 after $\chi = \mathsf{H}_2(com_1, \ldots, com_n)$ has already been queried, where Msg_i denotes the shares of $\{[w_j]\}_{j \in [1,m]}$ and $\{[z_j]\}_{j \in [1,N]}$ held by honest verifier $\mathcal{V}_i$ and r_i is sent to $\mathcal{V}_i$. If E_1 occurs, then when $\mathcal{A}$ makes a query $(com_1, \ldots, com_n)$ to obtain χ, com_i for some $i \in \mathcal{H}$ is either chosen by $\mathcal{A}$ without querying H_1 or output by H_1 for some query made by $\mathcal{A}$. In both cases, we have that $\Pr[E_1] \leq \frac{Q_1(t+1)}{2^\lambda} < \frac{Q_1 n}{2^\lambda}$, where $\mathcal{A}$ makes at most Q_1 queries to random oracle H_1.

Let E_2 be the event that the real-world execution outputs accept for $\mathcal{F}_{\mathsf{verifyprod}}$, but the ideal-world execution outputs abort when emulating $\mathcal{F}_{\mathsf{verifyprod}}$. Event E_2 happens if and only if there exists some $i \in [1, N]$ such that $z_i \neq x_i \cdot y_i$ but $z = \boldsymbol{x} \odot \boldsymbol{y}$ in the real protocol execution. We show that $\Pr[E_2 | \neg E_1] \leq \frac{(Q_2+1)(N-1)}{|\mathbb{K}|}$. Specifically, if E_1 does not occur, then the challenge χ is independent of the shares held by $t+1$ honest verifiers. These shares determine the secrets $w_1, \ldots, w_m$ on all circuit-input wires and the secrets $z_1, \ldots, z_N$ on the output wires of all multiplication gates. Furthermore, these secrets also determine the values on the output wires of all addition gates, which are computed locally. Overall, χ is independent of the secrets (x_i, y_i, z_i) of the i-th multiplication gate for all $i \in [1, N]$. If $\mathcal{A}$ does not make a query $(com_1, \ldots, com_n)$ to random oracle H_2, the behavior of $\mathcal{A}$ is independent of χ, i.e., $\chi = \mathsf{H}_2(com_1, \ldots, com_n)$ is independent from $z_i - x_i \cdot y_i$ for all $i \in [1, N]$. Since $\chi \in \mathbb{K}$ is uniformly random in the random-oracle model, the difference between checking $z = \boldsymbol{x} \odot \boldsymbol{y}$ and checking $z_i = x_i \cdot y_i$ for $i \in [1, N]$ is bounded by $\frac{N-1}{|\mathbb{K}|}$, according to the analysis in the proof of Theorem 1. If $\mathcal{A}$ queried $(com_1, \ldots, com_n)$ to H_2, the difference is bounded by $\frac{Q_2(N-1)}{|\mathbb{K}|}$, as $\mathcal{A}$ makes at most Q_2 queries to H_2. Overall, $\Pr[E_2 | \neg E_1] \leq \frac{(Q_2+1)(N-1)}{|\mathbb{K}|} < \frac{(Q_2+1)N}{2^\lambda}$, where $|\mathbb{K}| \geq 2^\lambda$. Therefore, we have

$$\begin{aligned}\Pr[E_2] &= \Pr[E_2 | E_1] \cdot \Pr[E_1] + \Pr[E_2 | \neg E_1] \cdot \Pr[\neg E_1] \\ &\leq \Pr[E_1] + \Pr[E_2 | \neg E_1] \leq \frac{Q_1 n + (Q_2+1)N}{2^\lambda}.\end{aligned}$$

In conclusion, for the verification phase of multiplication gates, the simulation of $\mathcal{S}$ is perfect, except with probability at most $\frac{Q_1 n+(Q_2+1)N}{2^\lambda}$.

Honest prover. In the verification phase of multiplication gates, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with $\mathcal{A}$ as follows:

1. After the circuit-evaluation phase, $\mathcal{S}$ has the shares of corrupted verifiers on the sharings $[w_1], \ldots, [w_m]$ and $[z_1], \ldots, [z_N]$ associated with the output wires of all circuit-input gates and multiplication gates.

2. $\mathcal{S}$ simulates the random oracles H_1 and H_2 by responding the queries with random values while keeping the consistency of answers.

3. On the behalf of the honest prover, $\mathcal{S}$ computes com_i honestly with the shares of corrupted verifier $\mathcal{V}_i$ and a uniform $r_i \in \{0,1\}^\lambda$ for each $i \in \mathcal{C}$, and samples $com_i \leftarrow \{0,1\}^\lambda$ for each $i \in \mathcal{H}$. Then, $\mathcal{S}$ sends $(com_1, \ldots, com_n)$ to $\mathcal{A}$ over an echo-broadcast channel.

4. Following the protocol specification, $\mathcal{S}$ computes the public coefficient $\chi \in \mathbb{K}$, and then computes the shares of $[\boldsymbol{x}], [\boldsymbol{y}], [z]$ held by corrupted verifiers. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$ by sending the shares of corrupted verifiers on $([\boldsymbol{x}], [\boldsymbol{y}], [z])$ to $\mathcal{A}$ and outputting accept to $\mathcal{A}$.

Below, we prove that $\mathcal{S}$ perfectly simulates the view of $\mathcal{A}$ in the verification phase of multiplication gates, except with probability at most $\frac{Q_1 n}{2^\lambda}$. The only difference between the real-world execution and ideal-world execution is that in the real-world execution, $\mathcal{A}$ receives the actual commitment com_i, while in the ideal-world execution, $\mathcal{A}$ always receives an independent random string in $\{0,1\}^\lambda$. The difference is bounded by the probability that $\mathcal{A}$ makes a query in the form of $(\cdots, r_i)$ to random oracle H_1 for some $i \in \mathcal{H}$, and is at most $\frac{Q_1(t+1)}{2^\lambda} < \frac{Q_1 n}{2^\lambda}$.

Combining the above proof for the verification phase of multiplication gates in the random-oracle model with the proof of Theorem 1 for the circuit-evaluation phase and the circuit-output verification phase, we conclude that the real-world execution is indistinguishable from the ideal-world execution, except with probability at most $\frac{Q_1 n+(Q_2+1)N}{2^\lambda}$. □

E Preprocess Circuits

We transform a general circuit C into another circuit C', which satisfies the following properties hold:

- For each input $\boldsymbol{w}$, we have that $C(\boldsymbol{w}) = C'(\boldsymbol{w})$.

- In the circuit-input layer, the number of circuit-input wires is the multiple of k. There are at least k circuit-output wires in the circuit-output layer. For the whole circuit, both of the number of addition gates and the number of multiplication gates are multiples of k.

- During the circuit evaluation, the gates with the same type (i.e., circuit-input gates, addition gates, multiplication gates and circuit-output gates) are divided into groups of k. Each group of k gates are evaluated simultaneously.

- **Circuit size:** $|C'| = |C| + O(k)$ where here $|C|$ denotes the number of all gates for a circuit C.

Circuit C' is obtained by adding new "dummy" circuit-input wires and circuit-output wires that take 0 as the values and inserting new "dummy" addition or multiplication gates that take these "dummy" circuit-input and circuit-output wires as their inputs and outputs. The detailed transformation from C to C' is described in Figure 16. In this figure, we also show how to divide k same-type gates into a group, and to store the

Procedure PrepCircuit

Inputs: Prover $\mathcal{P}$ and all verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ hold a circuit C defined over a field $\mathbb{F}$. Let M (resp., A) be the number of multiplication (resp., addition) gates in the circuit. Let k denote the number of secrets that are packed in a single sharing.

Circuit transformation: $\mathcal{P}$ and all verifiers transform a circuit C into another equivalent circuit C' as follows:

1. Insert $\lceil \frac{M}{k} \rceil \cdot k - M$ multiplication gates and $\lceil \frac{A}{k} \rceil \cdot k - A$ addition gates into the circuit. Specifically, each of these inserted gates takes two new circuit-input wires as the inputs of the gate, and creates one new circuit-output wire as its output. The values on these new circuit-input wires are set as 0.
2. Let h be the number of all circuit-output wires (including these new added circuit-output wires from the previous step). If $h < k$, insert $k - h$ new circuit-input wires that set 0 as the wire values, and then define these circuit-input wires as circuit-output wires directly.
3. Let m be the number of all circuit-input wires (including these new added circuit-input wires from the previous two steps). Insert $\lceil \frac{m}{k} \rceil \cdot k - m$ new circuit-input wires which set 0 as the wire values, and also define these circuit-input wires as circuit-output wires directly.

Pack wire values: Now, the number of gates that have the same type in the circuit is multiple of k. This is also the case for the number of circuit-input wires. Prover $\mathcal{P}$ and all verifiers do the following:

- For the circuit-input layer, divide the circuit-input wires into groups of k in an increasing order. For each group of k circuit-input wires, the corresponding input values will be stored in a single packed secret sharing.
- For all intermediate layers, following a predetermined topological order and the breadth-first search, execute the following:
  1. Collect k unused multiplication gates into a group. For the group of multiplication gates,
     - the values on the first input wires of these gates will be stored in a single packed secret sharing;
     - the values on the second input wires of these gates will be stored in a single packed secret sharing;
     - the values on the output wires of these gates will be stored in a single packed secret sharing.
  2. Collect k unused addition gates into a group. For the group of addition gates, divide the input and output values into packed secret sharings in the same way described as above.
- For the circuit-output layer, divide the single actual circuit-output wire along with any $k - 1$ dummy circuit-output wires into a group. The values on the group of circuit-output wires will be stored in a single packed secret sharing.

Figure 16: **Procedure for preprocessing a circuit without any communication.**

corresponding values in a single packed sharing. Note that the preprocessing procedure shown in Figure 16 only depends on the circuit C and does not need any communication.

From the construction of C', it is easy to prove that the above properties hold. In particular, for any input $\boldsymbol{w}$, we obtain $C(\boldsymbol{w}) = C'(\boldsymbol{w})$, as only "dummy" wires and gates are added and the input and output values are 0. When the state-of-the-art MPC protocol based on PSS [GPS21] is directly used to construct an interactive MVZK protocol, the compiled circuit C' has the size $|C'| = O(|C| + k \cdot d)$ where d is the depth of circuit C. We significantly optimize the size of circuit C' to $|C'| = |C| + O(k)$ by observing that the prover knows all wire values and thus it is unnecessary to evaluate the circuit layer-by-layer. Specifically, at most $k-1$ new addition gates and $k-1$ new multiplication gates are created, and there are at most $6(k-1)$ new circuit-input gates and $4(k-1)$ new circuit-output gates that are created. Our procedure for preprocessing circuits supports streaming an NIMVZK proof (i.e., proving a very large statement on-the-fly).

F Proof of Theorem 3

Theorem 9 (Theorem 3, restated)**.** *Let H_1 and H_2 be two random oracles. Protocol $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$ shown in Figures 6 and 7 securely realizes functionality $\mathcal{F}_{\mathsf{mvzk}}$ with soundness error at most $\frac{Q_1 n + (Q_2+1)(M+N)}{2^\lambda}$ in the $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$-hybrid model in the presence of a malicious adversary corrupting up to a prover and $t = d-k+1$ verifiers, where degree-d packed sharings are used in protocol $\Pi^{\mathsf{pss}}_{\mathsf{snimvzk}}$, each sharing packs k secrets, and Q_1 and Q_2 are the number of queries to H_1 and H_2 respectively.*

Proof. We first consider the case of a malicious prover (i.e., soundness), and then consider the case of an honest prover (i.e., zero knowledge), where exactly t verifiers are assumed to be corrupted in both cases according to Lemma 1. Let $n = 2d + 1$ be the number of all verifiers. In each case, we construct a PPT simulator $\mathcal{S}$, which is given access to functionality $\mathcal{F}_{\mathsf{mvzk}}$, and runs a PPT adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ for $\mathcal{A}$. In the simulation, whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.

Malicious prover. $\mathcal{S}$ simulates the random oracles H_1 and H_2 by responding the queries with random values while keeping the consistency of answers. $\mathcal{S}$ transforms the circuit C into an equivalent circuit C' following the protocol specification. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with $\mathcal{A}$ as follows:

- *Simulation of circuit evaluation phase:*

  1. From $i = 1$ to m, for the i-th group of k circuit-input wires, $\mathcal{S}$ receives the shares of $n - t = d + k$ honest verifiers from $\mathcal{A}$, and then reconstructs the *whole* sharing $[\boldsymbol{w}_i]$ from the shares of honest verifiers in $\mathcal{H}$. Then, $\mathcal{S}$ defines a witness $\bar{\boldsymbol{w}}$ using the values on *actual* circuit-input wires from $(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m)$, where the values on dummy circuit-input wires are ignored.
  2. For each group of k addition gates with whole input sharings $[\boldsymbol{x}]$ and $[\boldsymbol{y}]$, $\mathcal{S}$ computes the whole output sharing $[\boldsymbol{z}] := [\boldsymbol{x}] + [\boldsymbol{y}]$.
  3. From $i = 1$ to M, for the i-th group of k multiplication gates with whole input sharings $[\boldsymbol{x}_i]$ and $[\boldsymbol{y}_i]$, $\mathcal{S}$ receives the shares of $d + k$ honest verifiers from $\mathcal{A}$, and then reconstructs the whole output sharing $[\boldsymbol{z}_i]$ using the shares held by honest verifiers in $\mathcal{H}$.
  4. For each input sharing $[\boldsymbol{y}]$ such that $[\boldsymbol{y}]$ has not been generated, and y_j for $j \in [1, k]$ is identical to x_i for $i \in [1, k]$ stored in a different output sharing $[\boldsymbol{x}]$, $\mathcal{S}$ receives the shares of all honest verifiers from $\mathcal{A}$, and reconstructs the whole input sharing $[\boldsymbol{y}]$. Let $([\boldsymbol{x}_1], [\boldsymbol{y}_1], i_1, j_1), \ldots, ([\boldsymbol{x}_N], [\boldsymbol{y}_N], i_N, j_N)$ denote the wire tuples, where $\mathcal{S}$ knows the whole sharings. Let $[\boldsymbol{y}'_1], \ldots, [\boldsymbol{y}'_\ell]$ be the sharings $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_N]$ when the repetitive sharings are removed.

- *Simulation of Fiat-Shamir procedure and verification of multiplication tuples:*

1. $\mathcal{S}$ receives from $\mathcal{A}$ the public commitments $com_1, \ldots, com_n$. For $i \in \mathcal{H}$, $\mathcal{S}$ acts as honest verifier $\mathcal{V}_i$ and receives a randomness $r_i \in \{0,1\}^\lambda$ from $\mathcal{A}$.
2. For $i \in \mathcal{H}$, $\mathcal{S}$ uses the shares of $\{[\boldsymbol{w}_i]\}_{i\in[1,m]}$, $\{[\boldsymbol{z}_i]\}_{i\in[1,M]}$ and $\{[\boldsymbol{y}'_i]\}_{i\in[1,\ell]}$ held by honest verifier $\mathcal{V}_i$ as well as r_i to check the correctness of commitment com_i, where $\mathcal{S}$ makes the corresponding query to H_1 by itself. If the check fails, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.
3. $\mathcal{S}$ sets the public challenge $\chi := \mathsf{H}_2(com_1, \ldots, com_n)$ by making the corresponding query to H_2. Then, $\mathcal{S}$ computes the whole sharings $([\tilde{\boldsymbol{x}}_1], \ldots, [\tilde{\boldsymbol{x}}_M])$, $([\tilde{\boldsymbol{y}}_1], \ldots, [\tilde{\boldsymbol{y}}_M])$ and $[\tilde{\boldsymbol{z}}]$ following the protocol specification.
4. Let E_1 be the event that for some $i \in \mathcal{H}$, $com_i = \mathsf{H}_1(\mathsf{Msg}_i, r_i)$ and (Msg_i, r_i) was queried to H_1 after $\chi = \mathsf{H}_2(com_1, \ldots, com_n)$ has already been queried, where Msg_i consists of all the shares held by $\mathcal{V}_i$ and r_i is received by $\mathcal{V}_i$. If event E_1 occurs, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.
5. $\mathcal{S}$ emulates $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ by sending the whole sharings $\{[\tilde{\boldsymbol{x}}_i]\}_{i\in[1,M]}$, $\{[\tilde{\boldsymbol{y}}_i]\}_{i\in[1,M]}$ and $[\tilde{\boldsymbol{z}}]$ to $\mathcal{A}$. If there exists some $i \in [1, M]$ such that $\boldsymbol{z}_i \neq \boldsymbol{x}_i * \boldsymbol{y}_i$, then $\mathcal{S}$ sends abort to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{mvzk}}$, and aborts. Otherwise, $\mathcal{S}$ sends accept to $\mathcal{A}$.

- *Simulation of verification of wire tuples:* For each $i, j \in [1, k]$, let $([\boldsymbol{a}_1], [\boldsymbol{b}_1], i, j), \ldots, ([\boldsymbol{a}_{N'}], [\boldsymbol{b}_{N'}], i, j)$ denote the wire tuples in $L(i, j)$, and $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1. $\mathcal{S}$ receives the shares of honest verifiers in $\mathcal{H}$ on $[\boldsymbol{r}]$ and $[\boldsymbol{r}']$ from $\mathcal{A}$, and then reconstructs the whole sharings $[\boldsymbol{r}]$ and $[\boldsymbol{r}']$ using these shares.
2. $\mathcal{S}$ receives from $\mathcal{A}$ the public messages $(\boldsymbol{u}, \boldsymbol{u}')$, and then computes the whole sharings $[\boldsymbol{a}_0]$ and $[\boldsymbol{b}_0]$ following the protocol description.
3. Following the protocol specification, $\mathcal{S}$ computes $\alpha := \mathsf{H}_2(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ by making the corresponding query to H_2. Then, $\mathcal{S}$ computes the whole sharings $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$ using the whole sharings $\{[\boldsymbol{a}_h]\}_{h\in[0,N']}$, $\{[\boldsymbol{b}_h]\}_{h\in[0,N']}$ and public coefficient α.
4. On behalf of every honest verifier $\mathcal{V}_l$, $\mathcal{S}$ uses the shares of $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$ held by $\mathcal{V}_l$ to run the Open procedure with $\mathcal{A}$. If honest verifier $\mathcal{V}_l$ simulated by $\mathcal{S}$ aborts or $a_{h,i} \neq b_{h,j}$ for some $h \in [1, N']$, $\mathcal{S}$ sends abort_l to $\mathcal{F}_{\mathsf{mvzk}}$.

- *Simulation of circuit-output phase:* For the sharing on k circuit-output wires including the *actual* circuit-output wire, $\mathcal{S}$ plays the role of honest verifiers and runs the Open procedure with $\mathcal{A}$. Then, $\mathcal{S}$ sends $\bar{\boldsymbol{w}}$ to $\mathcal{F}_{\mathsf{mvzk}}$ who returns a decisional result $b \in \{\mathsf{true}, \mathsf{false}\}$ to $\mathcal{S}$. For each honest verifier $\mathcal{V}_i$ simulated by $\mathcal{S}$, if $\mathcal{V}_i$ accepts in the Open procedure, then $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}_{\mathsf{mvzk}}$, otherwise $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is indistinguishable from the ideal-world execution, except with probability at most $\frac{Q_1 n + (Q_2+1)(M+N)}{2^\lambda}$.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates the circuit-evaluation phase as described above. Specifically, $\mathcal{S}$ extracts a witness $\bar{\boldsymbol{w}}$ from $\mathcal{A}$, and also computes the whole sharing on each group of k input or output wires using the shares of $d + k$ honest verifiers.

It is clear that **Hybrid$_1$** has the identical distribution as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the Fiat-Shamir procedure as described above. Specifically, $\mathcal{S}$ checks the correctness of com_i for each $i \in \mathcal{H}$ following the protocol specification. In this procedure, $\mathcal{S}$ computes

$\chi = \mathsf{H}_2(com_1, \ldots, com_n)$, and also computes the whole sharings $([\tilde{\boldsymbol{x}}_1], \ldots, [\tilde{\boldsymbol{x}}_M])$, $([\tilde{\boldsymbol{y}}_1], \ldots, [\tilde{\boldsymbol{y}}_M])$ and $[\tilde{\boldsymbol{z}}]$ from $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i\in[1,M]}$ and χ. If event E_1 occurs, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{mvzk}}$, and then aborts.

The only difference between **Hybrid**$_1$ and **Hybrid**$_2$ is that if E_1 occurs, $\mathcal{S}$ does not abort in **Hybrid**$_1$, while $\mathcal{S}$ aborts in **Hybrid**$_2$. It is clear that $\Pr[E_1] \le \frac{Q_1(t+1)}{2^\lambda} < \frac{Q_1 n}{2^\lambda}$ following the proof of Theorem 2. Thus, the difference between **Hybrid**$_2$ and **Hybrid**$_1$ is bounded by $\frac{Q_1 n}{2^\lambda}$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ by checking that $\boldsymbol{z}_i = \boldsymbol{x}_i * \boldsymbol{y}_i$ for all $i \in [1, M]$ (instead of checking $\tilde{\boldsymbol{z}} = \sum_{h\in[1,M]} \tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h$) and sending the whole sharings $\{[\tilde{\boldsymbol{x}}_i]\}_{i\in[1,M]}$, $\{[\tilde{\boldsymbol{y}}_i]\}_{i\in[1,M]}$ and $[\tilde{\boldsymbol{z}}]$ to $\mathcal{A}$.

The only difference between **Hybrid**$_2$ and **Hybrid**$_3$ is that $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ checks whether $\tilde{\boldsymbol{z}} = \sum_{h\in[1,M]} \tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h$ in **Hybrid**$_2$, while $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ checks whether $\boldsymbol{z}_i = \boldsymbol{x}_i * \boldsymbol{y}_i$ for all $i \in [1, M]$ in **Hybrid**$_2$. If the event E_1 happens, both **Hybrid**$_2$ and **Hybrid**$_3$ abort. In the following, we assume that E_1 does not occur.

Let E_2 be the event that $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ outputs accept in **Hybrid**$_2$, but $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ outputs abort in **Hybrid**$_3$. Event E_2 occurs if and only if there exists some $i \in [1, M]$ such that $\boldsymbol{z}_i \ne \boldsymbol{x}_i * \boldsymbol{y}_i$ but $\tilde{\boldsymbol{z}} = \sum_{h\in[1,M]} \tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h$ holds. In the following, we prove that $\Pr[E_2] \le \frac{(Q_2+1)(M-1)}{2^\lambda}$. Specifically, since E_1 does not occur, the challenge χ is independent of the shares held by $d + k$ honest verifiers, which determine the secrets $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m$ on all circuit-input wires, $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_M$ on the output wires of all multiplication gates and $\boldsymbol{y}'_1, \ldots, \boldsymbol{y}'_\ell$ on the input sharings related to all wire tuples. Note that these secrets also determine the secrets on the output wires of all addition gates, which are computed locally. Overall, the challenge χ is independent of the secrets $\{(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{z}_i)\}_{i\in[1,M]}$ stored in all multiplication tuples. If $\mathcal{A}$ does not make a query $(com_1, \ldots, com_n)$ to random oracle H_2, coefficient $\chi = \mathsf{H}_2(com_1, \ldots, com_n)$ is independent from $(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{z}_i)$ for all $i \in [1, M]$. Consider the following two vectors of degree-$(M-1)$ polynomials over an extension field $\mathbb{K}$:

$$\boldsymbol{F}(X) = (\boldsymbol{x}_1 * \boldsymbol{y}_1) + (\boldsymbol{x}_2 * \boldsymbol{y}_2) \cdot X + \cdots + (\boldsymbol{x}_M * \boldsymbol{y}_M) \cdot X^{M-1},$$
$$\boldsymbol{G}(X) = \boldsymbol{z}_1 + \boldsymbol{z}_2 \cdot X + \cdots + \boldsymbol{z}_N \cdot X^{M-1}.$$

Therefore, we have that $\sum_{h\in[1,M]} \tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h = \boldsymbol{F}(\chi)$ and $\tilde{\boldsymbol{z}} = \boldsymbol{G}(\chi)$ for a uniform element $\chi \in \mathbb{K}$. If there exists some $i \in [1, M]$ such that $\boldsymbol{z}_i \ne \boldsymbol{x}_i * \boldsymbol{y}_i$, then $\boldsymbol{F}(X) - \boldsymbol{G}(X)$ is a non-zero polynomial vector of degree at most $(M-1)$. According to the Schwartz–Zippel lemma, for a random element $\chi \in \mathbb{K}$, the probability that $\tilde{\boldsymbol{x}}_h * \tilde{\boldsymbol{y}}_h - \tilde{\boldsymbol{z}} = \boldsymbol{F}(\chi) - \boldsymbol{G}(\chi) = 0^k$ is at most $\frac{M-1}{|\mathbb{K}|}$. Therefore, we directly obtain that $\Pr[E_2] \le \frac{M-1}{|\mathbb{K}|}$. If $\mathcal{A}$ queried $(com_1, \ldots, com_n)$ to H_2, we have $\Pr[E_2] \le \frac{Q_2(M-1)}{|\mathbb{K}|}$, where $\mathcal{A}$ makes at most Q_2 queries to random oracle H_2. Overall, we obtain $\Pr[E_2] \le \frac{(Q_2+1)(M-1)}{|\mathbb{K}|} \le \frac{(Q_2+1)(M-1)}{2^\lambda}$, where recall that $|\mathbb{K}| \ge 2^\lambda$. This means that the difference between **Hybrid**$_2$ and **Hybrid**$_3$ is bounded by $\frac{(Q_2+1)(M-1)}{2^\lambda}$.

Hybrid$_4$: In this hybrid, $\mathcal{S}$ simulates the verification of wire tuples as described above. In particular, for each $i, j \in [1, k]$, $\mathcal{S}$ checks that $a_{h,i} = b_{h,j}$ for all $h \in [1, N']$ rather than checking that $a_i = b_j$ for $\boldsymbol{a} = \sum_{h=0}^{N'} \alpha^h \cdot \boldsymbol{a}_h$ and $\boldsymbol{b} = \sum_{h=0}^{N'} \alpha^h \cdot \boldsymbol{b}_h$. $\mathcal{S}$ also reconstructs the whole sharings of $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$, and use the shares of honest verifiers to run the Open procedure with $\mathcal{A}$.

The only difference between **Hybrid**$_3$ and **Hybrid**$_4$ is that for $i, j \in [1, k]$, **Hybrid**$_3$ checks $a_i = b_j$, while **Hybrid**$_4$ checks $a_{h,i} = b_{h,j}$ for all $h \in [1, N']$. If the event E_1 (defined as above) occurs, $\mathcal{S}$ aborts in both of **Hybrid**$_3$ and **Hybrid**$_4$. In the following, we always assume that E_1 does not abort.

Let E_3 be the event that there exists some $h \in [1, N']$ such that $a_{h,i} \ne b_{h,j}$ but $a_i = b_j$ for some $i, j \in [1, k]$. Below, we prove that $\Pr[E_3] \le \frac{(Q_2+1)(N+1)}{2^\lambda}$. Let E_4 be the event that $\mathcal{A}$ first made a query $(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ to random oracle H_2, and either later $\mathcal{A}$ made a query $(com_1, \ldots, com_n)$ to H_2, or $\mathcal{A}$ did not query H_2 to obtain χ. Since $\mathcal{A}$ did not query H_2 to obtain χ before it makes a query $(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ to H_2, we have that the query $(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ happens with probability at most $\frac{Q_2}{|\mathbb{K}|}$. Thus, $\Pr[E_4] \leq \frac{Q_2}{|\mathbb{K}|} \leq \frac{Q_2}{2^\lambda}$. In the following analysis, we assume that E_4 does not occur.

If $\mathcal{A}$ did not make a query $(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ to obtain α. Then the behavior of $\mathcal{A}$ is independent of α. From $a_i = b_j$, we obtain that $\sum_{h=0}^{N'} \alpha^h \cdot a_{h,i} = \sum_{h=0}^{N'} \alpha^h \cdot b_{h,j}$. Consider the following polynomial of degree N' over a field $\mathbb{K}$:

$$H(X) = (a_{0,i} - b_{0,j}) + (a_{1,i} - b_{1,j}) \cdot X + \cdots + (a_{N',i} - b_{N',j}) \cdot X^{N'}.$$

Thus, we have that $H(\alpha) = \sum_{h=0}^{N'} \alpha^h \cdot a_{h,i} - \sum_{h=0}^{N'} \alpha^h \cdot b_{h,j} = a_i - b_j = 0$. If there exists some $h \in [1, N']$ such that $a_{h,i} \neq b_{h,j}$, H is a non-zero polynomial. According to the Schwartz–Zippel lemma, for a random element $\alpha \in \mathbb{K}$, the probability that $H(\alpha) = 0$ is at most $\frac{N'}{|\mathbb{K}|}$. If $\mathcal{A}$ queried $(\chi, \boldsymbol{u}, \boldsymbol{u}', i, j)$ to obtain α, then α is determined after $\chi, \boldsymbol{u}, \boldsymbol{u}'$ has been defined. The messages $\boldsymbol{u}, \boldsymbol{u}'$ determine the error $a_{0,i} - b_{0,j}$. Since both E_1 and E_4 do not happen, the challenge χ is determined after the secrets $\{a_{h,i}, b_{h,j}\}_{h \in [1,N']}$ have been defined. Overall, α is independent from the errors $\{a_{h,i} - b_{h,j}\}_{h \in [0,N']}$. Therefore, in this case, event E_3 happens with probability at most $\frac{Q_2 N'}{|\mathbb{K}|}$. In conclusion, we have that $\Pr[E_3 | \neg E_4] \leq \frac{(Q_2+1)N'}{|\mathbb{K}|} \leq \frac{(Q_2+1)N'}{2^\lambda}$. Further, we have the following:

$$\begin{aligned}\Pr[E_3] &= \Pr[E_3|E_4] \cdot \Pr[E_4] + \Pr[E_3|\neg E_4] \cdot \Pr[\neg E_4] \\ &\leq \Pr[E_4] + \Pr[E_3|\neg E_4] \leq \frac{(Q_2+1)N' + Q_2}{2^\lambda} < \frac{(Q_2+1)(N+1)}{2^\lambda}.\end{aligned}$$

Hence, we obtain that the difference between $\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$ is bounded by $\frac{(Q_2+1)(N+1)}{2^\lambda}$.

$\mathbf{Hybrid}_5$: In this hybrid, $\mathcal{S}$ simulates the circuit-output verification phase as described above. In particular, $\mathcal{S}$ sends $\bar{\boldsymbol{w}}$ to $\mathcal{F}_{\mathsf{mvzk}}$, and determines which honest verifiers obtain the output $b \in \{\mathsf{true}, \mathsf{false}\}$ relying on whether the honest verifier accepts or not during the Open procedure. This is the ideal-world execution.

If an honest verifier $\mathcal{V}_i$ aborts in the Open procedure, $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$ which outputs abort to $\mathcal{V}_i$ in $\mathbf{Hybrid}_5$. The verifier $\mathcal{V}_i$ has the same output (i.e., abort) in $\mathbf{Hybrid}_4$. If $\mathcal{V}_i$ does not abort, then it will obtain an output bit $\eta \in \{\mathsf{true}, \mathsf{false}\}$ computed following the protocol specification in $\mathbf{Hybrid}_4$, while it will receive $b \in \{\mathsf{true}, \mathsf{false}\}$ from functionality $\mathcal{F}_{\mathsf{mvzk}}$ in $\mathbf{Hybrid}_5$. Therefore, it is sufficient to bound the probability that $b \equiv \eta$. If an honest verifier $\mathcal{V}_i$ does not abort in $\mathbf{Hybrid}_4$, then all multiplication tuples and wire tuples are correct, and the circuit-output is also opened correctly. In addition, the evaluation of addition gates is trivially correct. Therefore, $\mathcal{V}_i$ will always obtain the correct output $\eta = C'(\bar{\boldsymbol{w}}) = C(\bar{\boldsymbol{w}})$ in $\mathbf{Hybrid}_4$, which has the identical distribution as b in $\mathbf{Hybrid}_5$.

Honest prover. If $\mathcal{S}$ receives false from functionality $\mathcal{F}_{\mathsf{mvzk}}$, then $\mathcal{S}$ aborts. $\mathcal{S}$ simulates random oracles H_1 and H_2 honestly. $\mathcal{S}$ transforms the circuit C into an equivalent circuit C' following the protocol specification. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verifyprod}}$, and interacts with $\mathcal{A}$ as follows:

- *Simulation of circuit evaluation phase:*
  1. For each group of k circuit-input wires with input vectors $\boldsymbol{w} \in \mathbb{F}^k$, $\mathcal{S}$ samples t uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on $[\boldsymbol{w}]$, and sends them to $\mathcal{A}$. Note that if $\boldsymbol{w} = 0^k$ corresponds to k dummy circuit-input wires, $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{w}]$ following the protocol description.
  2. For each group of k multiplication gates with an output vector $\boldsymbol{z} \in \mathbb{F}^k$, $\mathcal{S}$ samples t uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on the output packed sharing $[\boldsymbol{z}]$, and then sends them to $\mathcal{A}$.

3. For each input sharing $[\boldsymbol{y}]$ such that y_j is identical to x_i stored in a different output sharing $[\boldsymbol{x}]$ for $i, j \in [1, k]$, $\mathcal{S}$ samples t uniform elements in $\mathbb{F}$ as the shares of $[\boldsymbol{y}]$ held by corrupted verifiers, and then sends them to $\mathcal{A}$.

- *Simulation of Fiat-Shamir procedure and verification of multiplication tuples:*

1. On the behalf of the honest prover, $\mathcal{S}$ computes com_i honestly with the shares of corrupted verifier $\mathcal{V}_i$ and a uniform $r_i \in \{0,1\}^\lambda$ for each $i \in \mathcal{C}$, and samples $com_i \leftarrow \{0,1\}^\lambda$ for each $i \in \mathcal{H}$. Then, $\mathcal{S}$ sends $(com_1, \ldots, com_n)$ to $\mathcal{A}$ over an echo-broadcast channel.
2. Following the protocol specification, $\mathcal{S}$ computes a challenge $\chi \in \mathbb{K}$, and then computes the shares of $([\tilde{\boldsymbol{x}}_1], \ldots, [\tilde{\boldsymbol{x}}_M])$, $([\tilde{\boldsymbol{y}}_1], \ldots, [\tilde{\boldsymbol{y}}_M])$ and $[\tilde{\boldsymbol{z}}]$ held by corrupted verifiers. Then, $\mathcal{S}$ emulates $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ by sending these shares to $\mathcal{A}$ and outputting accept to $\mathcal{A}$.

- *Simulation of verification of wire tuples:* For each $i, j \in [1, k]$, let $([\boldsymbol{a}_1], [\boldsymbol{b}_1], i, j), \ldots, ([\boldsymbol{a}_{N'}], [\boldsymbol{b}_{N'}], i, j)$ denote the wire tuples in $L(i, j)$, and $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1. $\mathcal{S}$ samples $2t$ uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on $[\boldsymbol{r}]$ and $[\boldsymbol{r}']$, and then sends them to $\mathcal{A}$.
2. $\mathcal{S}$ samples $\boldsymbol{u}, \boldsymbol{u}' \leftarrow \mathbb{F}^k$, and then broadcasts $(\boldsymbol{u}, \boldsymbol{u}')$ to $\mathcal{A}$.
3. Following the protocol specification, $\mathcal{S}$ computes the shares of $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$ held by corrupted verifiers.
4. $\mathcal{S}$ samples k random elements in $\mathbb{K}$ as the shares of k honest verifiers on $[\boldsymbol{a}]$, and then uses the k elements along with the t shares of $[\boldsymbol{a}]$ held by corrupted verifiers to reconstruct the whole sharing $[\boldsymbol{a}]$. Besides, $\mathcal{S}$ samples $k-1$ uniform elements in $\mathbb{K}$, sets the secret $b_j = a_i$, and then uses the $k-1$ elements, b_j and the t shares of $[\boldsymbol{b}]$ held by corrupted verifiers to reconstruct the whole sharing $[\boldsymbol{b}]$.
5. On behalf of every honest verifier $\mathcal{V}_i$, $\mathcal{S}$ uses the shares of $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$ held by $\mathcal{V}_i$ to run the Open procedure with $\mathcal{A}$. If $\mathcal{V}_i$ aborts, $\mathcal{S}$ sends abort_i to functionality $\mathcal{F}_{\mathsf{mvzk}}$.

- *Simulation of circuit-output phase:*

1. $\mathcal{S}$ uses the shares of output sharing $[\boldsymbol{z}]$ held by t corrupted verifiers on k circuit-output gates (involving the actual circuit-output wire) along with k zero secrets to compute the shares of honest verifiers on $[\boldsymbol{z}]$.
2. $\mathcal{S}$ uses the shares of $[\boldsymbol{z}]$ held by honest verifiers to run the Open procedure with $\mathcal{A}$. For each $i \in \mathcal{H}$, if $\mathcal{V}_i$ accepts in the Open procedure, then $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}_{\mathsf{mvzk}}$, otherwise $\mathcal{S}$ sends abort_i to $\mathcal{F}_{\mathsf{mvzk}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is indistinguishable from the ideal-world execution, except with probability at most $\frac{Q_1 n}{2^\lambda}$.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates the Fiat-Shamir procedure and the verification phase of multiplication tuples as described above. In particular, $\mathcal{S}$ samples $com_i \leftarrow \{0,1\}^\lambda$ for each $i \in \mathcal{H}$ and emulates $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ by sending the shares of corrupted verifiers on $(\{[\tilde{\boldsymbol{x}}_i], [\tilde{\boldsymbol{y}}_i]\}_{i \in [1,M]}, [\tilde{\boldsymbol{z}}])$ to $\mathcal{A}$.

It is clear that the shares of $(\{[\tilde{\boldsymbol{x}}_i], [\tilde{\boldsymbol{y}}_i]\}_{i \in [1,M]}, [\tilde{\boldsymbol{z}}])$ held by corrupted verifiers are computed following the protocol description in **Hybrid$_1$**, and thus have the identical distribution as that in **Hybrid$_0$**. The only difference between **Hybrid$_0$** and **Hybrid$_1$** is that for each $i \in \mathcal{H}$, **Hybrid$_0$** computes com_i with the shares of honest verifier $\mathcal{V}_i$, while **Hybrid$_1$** samples com_i uniformly at random. Let E_5 be the event that $\mathcal{A}$ makes a query $(\hat{w}^i_1, \ldots, \hat{w}^i_m, \hat{z}^i_1, \ldots, \hat{z}^i_M, \hat{y}^i_1, \ldots, \hat{y}^i_\ell, r_i)$ to random oracle H_1 for some $i \in \mathcal{H}$. It is sufficient to show the probability that E_5 occurs. For a fixed index $i \in \mathcal{H}$, the probability that E_5 happens is at most $\frac{Q_1}{2^\lambda}$. There are at most $n - t$ honest verifiers in $\mathcal{H}$, and thus we have that $\Pr[E_5] \le \frac{Q_1(n-t)}{2^\lambda} < \frac{Q_1 n}{2^\lambda}$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the verification phase of wire tuples as described above. Specifically, for each $i, j \in [1, k]$, $\mathcal{S}$ samples random elements as the shares of $[\boldsymbol{r}]$ and $[\boldsymbol{r}']$ held by corrupted verifiers, and samples $(\boldsymbol{u}, \boldsymbol{u}')$ uniformly at random. Additionally, $\mathcal{S}$ samples k random elements and $k-1$ random elements as the shares of $[\boldsymbol{a}]$ and $[\boldsymbol{b}]$ held by a part of honest verifiers, when guaranteeing $a_i = b_j$.

According to the definition of packed secret sharings, the shares of $[\boldsymbol{r}]$ and $[\boldsymbol{r}']$ held by corrupted verifiers are uniform in **Hybrid$_1$**. Note that the secret vectors $\boldsymbol{r}$ and $\boldsymbol{r}'$ are perfectly hidden against the adversary's view. Thus, $\boldsymbol{u}$ and $\boldsymbol{u}'$ are uniformly distributed in **Hybrid$_1$**. Due to that $\boldsymbol{a}_0, \boldsymbol{b}_0 \in \mathbb{K}^k$ are uniformly random except for $a_{0,i} = b_{0,j}$, $\boldsymbol{a}$ and $\boldsymbol{b}$ are uniform in $\mathbb{K}^k$ such that $a_i = b_j$ in **Hybrid$_1$**. Overall, we have that **Hybrid$_2$** has the identical distribution as **Hybrid$_1$**.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the circuit-output verification phase as described above. In particular, $\mathcal{S}$ uses the shares of t corrupted verifiers along with k zero secrets to reconstruct the whole sharing $[\boldsymbol{z}]$ on the k circuit-output wires. Then, $\mathcal{S}$ runs the Open procedure with $\mathcal{A}$ using the shares of honest verifiers on $[\boldsymbol{z}]$. For $i \in \mathcal{H}$, $\mathcal{S}$ sends either $\mathsf{continue}_i$ or abort_i to functionality $\mathcal{F}_{\mathsf{mvzk}}$ depending on whether $\mathcal{V}_i$ accepts or not in the Open procedure.

In **Hybrid$_2$**, the circuit is evaluated correctly in the case of an honest prover. Thus, the corrupted verifiers will obtain a circuit-output $C'(\bar{\boldsymbol{w}}) = C(\bar{\boldsymbol{w}}) = 0$. In **Hybrid$_3$**, $\mathcal{S}$ reconstructs the whole sharing $[\boldsymbol{z}]$ such that $\boldsymbol{z} = 0^k$. Therefore, the simulation of the Open procedure in **Hybrid$_3$** has the identical distribution as that in **Hybrid$_2$**. Note that the outputs of honest verifiers in **Hybrid$_3$** have the identical distribution as that in **Hybrid$_2$**. Thus, **Hybrid$_3$** is perfectly indistinguishable from **Hybrid$_2$**.

Hybrid$_4$: In this hybrid, $\mathcal{S}$ simulates the circuit evaluation phase as described above. Specifically, $\mathcal{S}$ samples uniform elements in $\mathbb{F}$ as the shares of corrupted verifiers on the packed sharings that will be sent by the honest prover in this phase. This is the ideal-world execution.

In **Hybrid$_3$**, $\mathcal{S}$ uses the actual wire values and the Share procedure to generate the packed sharings, while in **Hybrid$_4$**, $\mathcal{S}$ simply samples uniform elements as the shares of corrupted verifiers. According to the definition of packed secret sharings, the distribution of the shares of corrupted verifiers in **Hybrid$_4$** is identical to that in **Hybrid$_3$**. Therefore, **Hybrid$_4$** has the identical distribution as **Hybrid$_3$**.

In conclusion, the real-world execution is indistinguishable from the ideal-world execution except with probability at most $\frac{Q_1 n + (Q_2+1)(M+N)}{2^\lambda}$, which completes the proof. □

G Proof of Theorem 4

Theorem 10 (Theorem 4, restated)**.** *Let* $\mathsf{H}_2 : \{0,1\}^* \to \mathbb{K}$ *be a random oracle. Protocol* $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ *shown in Figure 8 securely realizes functionality* $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ *with soundness error at most* $\frac{4\lceil \log M \rceil + 5Q_2}{2^\lambda - 3}$ *in the presence of a malicious adversary corrupting up to the prover and exactly t verifiers, where Q_2 is the number of queries to random oracle* H_2.

Proof. For any PPT adversary $\mathcal{A}$, we construct a PPT simulator $\mathcal{S}$, which is given access to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ and runs $\mathcal{A}$ as a subroutine. Whenever $\mathcal{A}$ aborts, $\mathcal{S}$ sends abort to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ and also aborts.

Malicious prover. $\mathcal{S}$ receives the whole sharings $([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_M])$, $([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_M])$ and $[\boldsymbol{z}]$ from functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, and computes $\boldsymbol{d} := \boldsymbol{z} - \sum_{h \in [1,M]} \boldsymbol{x}_h * \boldsymbol{y}_h \in \mathbb{K}^k$. Then, $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

- *Simulation of sub-protocol* $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$: Let $([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_\ell])$ and $([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_\ell])$ be the input packed sharings. Specifically, $\mathcal{S}$ receives the shares of $[\boldsymbol{r}]$ held by honest verifiers in $\mathcal{H}$ from $\mathcal{A}$, and reconstructs the whole sharing $[\boldsymbol{r}]$. Then, $\mathcal{S}$ receives $\boldsymbol{u}$ from $\mathcal{A}$, and computes the whole sharing $[\boldsymbol{z}] = \boldsymbol{u} - [\boldsymbol{r}]$.

- *Simulation of sub-protocol* $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$: For each $i \in [1, m]$, let $(([\boldsymbol{x}_{i,1}], \dots, [\boldsymbol{x}_{i,\ell}]), ([\boldsymbol{y}_{i,1}], \dots, [\boldsymbol{y}_{i,\ell}]), [\boldsymbol{z}_i])$ be the input packed inner-product tuple. By interacting with $\mathcal{A}$, $\mathcal{S}$ simulates the protocol execution as follows:

  1. For $j \in [1, \ell]$, $\mathcal{S}$ computes the whole sharings $[\boldsymbol{f}_j(\cdot)]$ and $[\boldsymbol{g}_j(\cdot)]$ from the whole input sharings $\{[\boldsymbol{x}_{i,j}]\}_{i\in[1,m]}$ and $\{[\boldsymbol{y}_{i,j}]\}_{i\in[1,m]}$.
  2. For $i \in [m + 1, 2m - 1]$, $\mathcal{S}$ simulates the execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input sharings $([\boldsymbol{f}_1(i)], \dots, [\boldsymbol{f}_\ell(i)])$ and $([\boldsymbol{g}_1(i)], \dots, [\boldsymbol{g}_\ell(i)])$, and obtains the whole sharing $[\boldsymbol{z}_i]$.
  3. $\mathcal{S}$ computes the whole sharing $[\boldsymbol{h}(\cdot)]$ from the whole sharings $[\boldsymbol{z}_1], \dots, [\boldsymbol{z}_{2m-1}]$.
  4. $\mathcal{S}$ computes a public coefficient α following the protocol specification. If $\alpha \in [1, m]$, then $\mathcal{S}$ sends abort to functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ and aborts. Otherwise, $\mathcal{S}$ computes the whole sharings $[\boldsymbol{f}_j(\alpha)]$ and $[\boldsymbol{g}_j(\alpha)]$ for $j \in [1, \ell]$, and also computes the whole sharing $[\boldsymbol{h}(\alpha)]$.

- *Simulation of dimension-reduction*: Following the protocol specification, $\mathcal{S}$ simulates the view of $\mathcal{A}$ in the iterative manner. For the current iteration, let $(([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ be the input packed inner-product tuple, where $m > 2$.

  1. For each $i \in [1, 2]$, $\mathcal{S}$ computes the whole sharings $([\boldsymbol{a}_{i,1}], \dots, [\boldsymbol{a}_{i,\ell}])$ and $([\boldsymbol{b}_{i,1}], \dots, [\boldsymbol{b}_{i,\ell}])$ from the input inner-product tuple, where $\ell = m/2$.
  2. $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input sharings $([\boldsymbol{a}_{1,1}], \dots, [\boldsymbol{a}_{1,\ell}])$ and $([\boldsymbol{b}_{1,1}], \dots, [\boldsymbol{b}_{1,\ell}])$, and then obtains the whole sharing $[\boldsymbol{c}_1]$.
  3. $\mathcal{S}$ computes the whole sharing $[\boldsymbol{c}_2] = [\boldsymbol{z}] - [\boldsymbol{c}_1]$.
  4. $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ as described above for input sharings $([\boldsymbol{a}_{i,1}], \dots, [\boldsymbol{a}_{i,\ell}])$, $([\boldsymbol{b}_{i,1}], \dots, [\boldsymbol{b}_{i,\ell}])$ and $[\boldsymbol{c}_i]$ for $i \in [1, 2]$. Then, $\mathcal{S}$ uses the whole sharings $([\boldsymbol{a}_1], \dots, [\boldsymbol{a}_\ell])$, $([\boldsymbol{b}_1], \dots, [\boldsymbol{b}_\ell])$ and $[\boldsymbol{c}]$ output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ to update $([\boldsymbol{x}_1], \dots, [\boldsymbol{x}_{m/2}])$, $([\boldsymbol{y}_1], \dots, [\boldsymbol{y}_{m/2}])$ and $[\boldsymbol{z}]$ respectively that are the input packed sharings used in the next iteration.
  5. $\mathcal{S}$ update γ as α output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$, and also set $m := m/2$. Then, $\mathcal{S}$ performs the next iteration.

- *Simulation of randomization*: $\mathcal{S}$ has the whole sharings $([\boldsymbol{x}_1], [\boldsymbol{x}_2])$, $([\boldsymbol{y}_1], [\boldsymbol{y}_2])$ and $[\boldsymbol{z}]$ output in the dimension-reduction phase. $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

  1. $\mathcal{S}$ receives the shares of $([\boldsymbol{x}_0], [\boldsymbol{y}_0])$ held by honest verifiers in $\mathcal{H}$ from $\mathcal{A}$, and then reconstructs the whole sharings $[\boldsymbol{x}_0]$ and $[\boldsymbol{y}_0]$ using these shares.
  2. For $i \in [0, 1]$, $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input $([\boldsymbol{x}_i], [\boldsymbol{y}_i])$, and then obtains the whole sharing $[\boldsymbol{z}_i]$.
  3. $\mathcal{S}$ computes the whole sharing $[\boldsymbol{z}_2] := [\boldsymbol{z}] - [\boldsymbol{z}_1]$.
  4. $\mathcal{S}$ simulates the execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ as described above for input sharings $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i\in[0,2]}$, and then obtains the whole sharings $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$.
  5. $\mathcal{S}$ receives an output either abort or accept from functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$. $\mathcal{S}$ uses the shares of honest verifiers on $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ to run the Open procedure with $\mathcal{A}$. For every honest verifier $\mathcal{V}_i$ (simulated by $\mathcal{S}$), if $\mathcal{V}_i$ aborts in the Open procedure, then $\mathcal{S}$ sends abort_i to functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, otherwise $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is indistinguishable from the ideal-world execution, except with probability at most $\frac{4\lceil \log M \rceil + 5Q_2}{2^{\lambda-3}}$.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates two sub-protocols $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ and $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ as described above.

In **Hybrid$_1$**, $\mathcal{S}$ only reconstructs the whole sharings from the shares of honest verifiers in $\mathcal{H}$. This does not change the behaviors of honest verifiers. Thus, **Hybrid$_1$** has the identical distribution as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the dimension-reduction phase as described above.

In the dimension-reduction phase of **Hybrid$_2$**, $\mathcal{S}$ still simply reconstructs the whole sharings from the shares of honest verifiers in $\mathcal{H}$. Therefore, the distributions of **Hybrid$_1$** and **Hybrid$_2$** are identical.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the randomization phase as described above. In particular, $\mathcal{S}$ reconstructs the whole sharings $[\boldsymbol{a}]$, $[\boldsymbol{b}]$ and $[\boldsymbol{c}]$. Then, $\mathcal{S}$ uses the shares of honest verifiers on $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ to run the Open procedure with $\mathcal{A}$. For every honest verifier $\mathcal{V}_i$, if $\mathcal{V}_i$ aborts in the Open procedure, then $\mathcal{S}$ sends abort_i to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, which sends abort to $\mathcal{V}_i$. Otherwise, $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, which sends the decision result whether $\boldsymbol{z} = \sum_{h\in[1,M]} \boldsymbol{x}_h * \boldsymbol{y}_h$ or not to $\mathcal{V}_i$, where $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_M]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_M]), [\boldsymbol{z}])$ is the input inner-product tuple. This is the ideal-world execution.

Clearly, the simulation of the Open procedure is perfect. Let E_1 be the event that the input packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_M]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_M]), [\boldsymbol{z}])$ is *incorrect* and the protocol execution does not abort, but some honest verifier outputs accept in **Hybrid$_2$**, while the honest verifier outputs abort in **Hybrid$_3$**. The only difference between **Hybrid$_2$** and **Hybrid$_3$** is that E_1 occurs. Below, we prove $\Pr[E_1] \le \frac{4\lceil \log M\rceil + 5Q_2}{2^\lambda - 3}$.

Let $\sigma = \lceil \log M\rceil$. We divide the protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$ into σ iterations, where the first $\sigma-1$ iterations execute in the dimension-reduction phase and the σ-th iteration is run in the randomization phase. Let $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_\sigma$ be the queries to random oracle H_2 during the σ iterations, which should be made to generate the challenges. Let E_2 be the event that both of the following hold:

- For any $i, j \in [1, \sigma]$ with $i < j$, if $\mathcal{A}$ queries both $\boldsymbol{q}_i$ and $\boldsymbol{q}_j$ to random oracle H_2, then $\boldsymbol{q}_i$ was queried first.
- For any $i \in [1, \sigma]$ where the packed inner-product tuple output in the $(i-1)$-th iteration is *incorrect* but the packed inner-product tuple output in the i-th iteration is *correct*, then $\mathcal{A}$ queries $\boldsymbol{q}_i$ to random oracle H_2. Here the tuple in the 0-th iteration is defined as the packed inner-product tuple input by all verifiers.

Suppose that E_1 occurs and E_2 does not. Then, at least one of the following cases holds:

1. For some $i < j$ with $i, j \in [1, \sigma]$, $\mathcal{A}$ queried $\boldsymbol{q}_j$, and either later made a query $\boldsymbol{q}_i$ or did not query $\boldsymbol{q}_i$.
2. There exists some $i \in [1, \sigma]$ where the packed inner-product tuple output in the $(i-1)$-th iteration is *incorrect* but the tuple output in the i-th iteration is *correct*, and $\mathcal{A}$ did not query $\boldsymbol{q}_i$.

If the first case occurs, let $i, j \in [1, \sigma]$ be such a pair of indices such that $i < j$, $j - i$ is smallest, and $\mathcal{A}$ queried $\boldsymbol{q}_j$ and either later queried $\boldsymbol{q}_i$ or did not query $\boldsymbol{q}_i$. Therefore, $\mathcal{A}$ did not make a query $\boldsymbol{q}_{j-1}$ to random oracle H_2 before it queries $\boldsymbol{q}_j$ to H_2. One of the values in $\boldsymbol{q}_j$ is $\alpha_{j-1} = \mathsf{H}_2(\boldsymbol{q}_{j-1}) \in \mathbb{K}$, and thus the behavior of $\mathcal{A}$ up until querying $\boldsymbol{q}_j$ is independent of coefficient α_{j-1}. So the query $\boldsymbol{q}_j$ occurs with probability at most $\frac{Q_2}{|\mathbb{K}|} \le \frac{Q_2}{2^\lambda}$, where $|\mathbb{K}| \ge 2^\lambda$.

Suppose that the second case happens, let $i \in [1, \sigma]$ be the first index such that the packed inner-product tuple output in the $(i-1)$-th iteration is *incorrect*, the packed inner-product tuple output in the i-th iteration is *correct* and $\mathcal{A}$ did not query $\boldsymbol{q}_i$. Before giving the probability that the second case occurs, we prove the following two lemmas.

Lemma 5. *Suppose that $\mathcal{A}$ does not make a query of random oracle H_2 to obtain the coefficient α in the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$. If there exists at least one incorrect packed inner-product tuple and protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ shown in Figure 10 does not abort, then the resulting inner-product tuple output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ is correct with probability at most $\frac{2(m-1)}{|\mathbb{K}|-m}$.*

Proof. If $\alpha \in [1, m]$, then the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ aborts. Therefore, there are at most $|\mathbb{K}| - m$ choices of α that do not cause an abort of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$. In the following, we assume that $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ does not abort. Since there are at least one incorrect packed inner-product tuple, we have that $\sum_{j\in[1,\ell]} \boldsymbol{f}_j * \boldsymbol{g}_j \neq \boldsymbol{h}$. Note that the polynomial $\sum_{j\in[1,\ell]} \boldsymbol{f}_j * \boldsymbol{g}_j - \boldsymbol{h}$ has a degree at most $2(m-1)$. Since $\mathcal{A}$ does not make a H_2-query $(\gamma, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, \boldsymbol{u}_{m+1}, \ldots, \boldsymbol{u}_{2m-1})$ to obtain α, the behavior of $\mathcal{A}$ is independent from α, meaning that α is independent of the polynomial vectors $\{\boldsymbol{f}_j, \boldsymbol{g}_j\}_{j\in[1,\ell]}$ and $\boldsymbol{h}$. According to the Schwartz–Zippel lemma, for a uniform element $\alpha \in \mathbb{K}\backslash[1, m]$, the probability that $\sum_{j\in[1,\ell]} \boldsymbol{f}_j * \boldsymbol{g}_j - \boldsymbol{h} = 0^k$ is bounded by $\frac{2(m-1)}{|\mathbb{K}|-m}$. Therefore, if protocol $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ does not abort, it outputs a correct packed inner-product tuple with probability at most $\frac{2(m-1)}{|\mathbb{K}|-m}$, which completes the proof. □

Lemma 6. *For each iteration of protocol $\Pi^{\mathsf{pss}}_{\mathsf{verifyprod}}$, if the input packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ is incorrect, the output packed inner-product tuple $(([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\ell]), ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\ell]), [\boldsymbol{c}])$ is correct with probability at most $\frac{4}{|\mathbb{K}|-3}$, where $\ell = m/2$.*

Proof. For the i-th iteration with $i \in [1, \sigma - 1]$ in the dimension-reduction phase, suppose that the input inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ is incorrect. We first show that at least one of the following two packed inner-product tuples is incorrect:

$$(([\boldsymbol{a}_{i,1}], \ldots, [\boldsymbol{a}_{i,\ell}]), ([\boldsymbol{b}_{i,1}], \ldots, [\boldsymbol{b}_{i,\ell}]), [\boldsymbol{c}_i]) \text{ for } i \in [1, 2],$$

where for $i \in [1, 2]$, $([\boldsymbol{a}_{i,1}], \ldots, [\boldsymbol{a}_{i,\ell}]) = ([\boldsymbol{x}_{(i-1)\ell+1}], \ldots, [\boldsymbol{x}_{(i-1)\ell+\ell}])$, $([\boldsymbol{b}_{i,1}], \ldots, [\boldsymbol{b}_{i,\ell}]) = ([\boldsymbol{y}_{(i-1)\ell+1}], \ldots, [\boldsymbol{y}_{(i-1)\ell+\ell}])$ and $[\boldsymbol{c}_1] + [\boldsymbol{c}_2] = [\boldsymbol{z}]$. If the first packed inner-product tuple is incorrect, then the statement holds. Otherwise, we have that $\sum_{h\in[1,\ell]} \boldsymbol{a}_{1,h} * \boldsymbol{b}_{1,h} = \boldsymbol{c}_1$. As the input packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ is incorrect, we have that $\sum_{h\in[1,m]} \boldsymbol{x}_h * \boldsymbol{y}_h \neq \boldsymbol{z}$. Thus, we have the following:

$$\sum_{h\in[1,\ell]} \boldsymbol{a}_{2,h} * \boldsymbol{b}_{2,h} = \sum_{h\in[1,m]} \boldsymbol{x}_h * \boldsymbol{y}_h - \sum_{h\in[1,\ell]} \boldsymbol{a}_{1,h} * \boldsymbol{b}_{1,h} = \sum_{h\in[1,m]} \boldsymbol{x}_h * \boldsymbol{y}_h - \boldsymbol{c}_1 \neq \boldsymbol{z} - \boldsymbol{c}_1 = \boldsymbol{c}_2.$$

This means that the second packed inner-product tuple $(([\boldsymbol{a}_{2,1}], \ldots, [\boldsymbol{a}_{2,\ell}]), ([\boldsymbol{b}_{2,1}], \ldots, [\boldsymbol{b}_{2,\ell}]), [\boldsymbol{c}_2])$ is incorrect. According to Lemma 5, we obtain that the resulting packed inner-product tuple $(([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\ell]), ([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\ell]), [\boldsymbol{c}])$ output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ is correct with probability at most $\frac{2}{|\mathbb{K}|-2}$.

For the σ-th iteration (i.e., the randomization phase), suppose that the input packed inner-product tuple $(([\boldsymbol{x}_1], [\boldsymbol{x}_2]), ([\boldsymbol{y}_1], [\boldsymbol{y}_2]), [\boldsymbol{z}])$ is incorrect. In the following, let $\{([\boldsymbol{x}_i], [\boldsymbol{y}_i], [\boldsymbol{z}_i])\}_{i\in[0,2]}$ be three multiplication tuples such that $[\boldsymbol{z}_1] + [\boldsymbol{z}_2] = [\boldsymbol{z}]$. Following a similar analysis as described above, we have that at least one of two multiplication tuples $([\boldsymbol{x}_1], [\boldsymbol{y}_1], [\boldsymbol{z}_1])$ and $([\boldsymbol{x}_2], [\boldsymbol{y}_2], [\boldsymbol{z}_2])$ is incorrect. No matter what correctness of the additional multiplication tuple $([\boldsymbol{x}_0], [\boldsymbol{y}_0], [\boldsymbol{z}_0])$ is, the resulting multiplication tuple $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ is correct with probability at most $\frac{4}{|\mathbb{K}|-3}$, according to Lemma 5.

In conclusion, the statement described in this lemma holds with probability at most $\frac{4}{|\mathbb{K}|-3}$. □

According to Lemma 6, we obtain the probability that the second case occurs is bounded by $\frac{4\lceil \log M\rceil}{|\mathbb{K}|-3} \leq \frac{4\lceil \log M\rceil}{2^\lambda-3}$, by taking a union bound over all $\sigma = \lceil \log M\rceil$ iterations. Overall, we have $\Pr[E_1|\neg E_2] \leq \frac{Q_2}{2^\lambda} + \frac{4\lceil \log M\rceil}{2^\lambda-3} < \frac{Q_2+4\lceil \log M\rceil}{2^\lambda-3}$.

Below, suppose that both E_1 and E_2 occur. For some $i \in [1, \sigma]$, in the i-th iteration, $\mathcal{A}$ made the queries $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_i$ to random oracle H_2 in the increasing order. Note that the entire partial transcript up to the messages sent in the i-th iteration is fixed, at the time when $\mathcal{A}$ queries $\boldsymbol{q}_i$ to obtain α_i. Furthermore, the input coefficient χ, which is determined after the secrets of the input packed inner-product tuple have been defined, is involved in the first query $\boldsymbol{q}_1$. Therefore, the coefficient α_i used in the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ for the i-th iteration is determined after the secret polynomials $(\boldsymbol{f}_1(\cdot), \ldots, \boldsymbol{f}_\ell(\cdot))$, $(\boldsymbol{g}_1(\cdot), \ldots, \boldsymbol{g}_\ell(\cdot))$ and $\boldsymbol{h}(\cdot)$ have been defined. Based on Lemma 6, we have $\Pr[E_1|E_2] \leq \frac{4Q_2}{|\mathbb{K}|-3} \leq \frac{4Q_2}{2^\lambda-3}$.

We can sum up the bounds as described above to get

$$\begin{aligned}\Pr[E_1] &= \Pr[E_1|E_2] \cdot \Pr[E_2] + \Pr[E_1|\neg E_2] \cdot \Pr[\neg E_2] \\ &\leq \Pr[E_1|E_2] + \Pr[E_1|\neg E_2] \leq \frac{4\lceil \log M \rceil + 5Q_2}{2^\lambda - 3}.\end{aligned}$$

Honest prover. $\mathcal{S}$ receives the shares of corrupted verifiers on $([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_M])$, $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_M])$ and $[\boldsymbol{z}]$ from $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$. If $\mathcal{S}$ receives abort from $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, it aborts. Then, $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

- *Simulation of sub-protocol* $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$: Let $([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_\ell])$ and $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_\ell])$ be the input packed sharings. $\mathcal{S}$ samples random elements in $\mathbb{K}$ as the shares of $[\boldsymbol{r}]$ held by corrupted verifiers, and sends them to $\mathcal{A}$. Then, $\mathcal{S}$ samples $\boldsymbol{u} \leftarrow \mathbb{K}^k$, and broadcasts $\boldsymbol{u}$ to $\mathcal{A}$. Following the protocol description, $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{z}] = [\sum_{h\in[1,\ell]} \boldsymbol{x}_h * \boldsymbol{y}_h]$.

- *Simulation of sub-protocol* $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$: For each $i \in [1, m]$, let $(([\boldsymbol{x}_{i,1}], \ldots, [\boldsymbol{x}_{i,\ell}]), ([\boldsymbol{y}_{i,1}], \ldots, [\boldsymbol{y}_{i,\ell}]), [\boldsymbol{z}_i])$ be the input packed inner-product tuple. $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

  1. For $j \in [1, \ell]$, $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{f}_j(\cdot)]$ and $[\boldsymbol{g}_j(\cdot)]$ from their shares of $\{[\boldsymbol{x}_{i,j}]\}_{i\in[1,m]}$ and $\{[\boldsymbol{y}_{i,j}]\}_{i\in[1,m]}$.
  2. For $i \in [m + 1, 2m - 1]$, $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input sharings $([\boldsymbol{f}_1(i)], \ldots, [\boldsymbol{f}_\ell(i)])$ and $([\boldsymbol{g}_1(i)], \ldots, [\boldsymbol{g}_\ell(i)])$, and obtains the shares of $[\boldsymbol{z}_i]$ held by corrupted verifiers.
  3. $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{h}(\cdot)]$ from their shares of $\{[\boldsymbol{z}_i]\}_{i\in[1,2m-1]}$.
  4. $\mathcal{S}$ computes the coefficient $\alpha \in \mathbb{K}$ following the protocol specification. If $\alpha \in [1, m]$, $\mathcal{S}$ sends abort to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$ and aborts. Otherwise, $\mathcal{S}$ computes the shares of corrupted verifiers on $\{[\boldsymbol{f}_j(\alpha)], [\boldsymbol{g}_j(\alpha)]\}_{j\in[1,\ell]}$ and $[\boldsymbol{h}(\alpha)]$.

- *Simulation of dimension-reduction*: Simulator $\mathcal{S}$ simulates the view of adversary $\mathcal{A}$ in the following iterative manner. For the current iteration, let $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_m]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_m]), [\boldsymbol{z}])$ be the input packed inner-product tuple, where $m > 2$.

  1. For $i \in [1, 2]$, $\mathcal{S}$ computes the shares of $([\boldsymbol{a}_{i,1}], \ldots, [\boldsymbol{a}_{i,\ell}])$ and $([\boldsymbol{b}_{i,1}], \ldots, [\boldsymbol{b}_{i,\ell}])$ held by corrupted verifiers from their shares of $\{([\boldsymbol{x}_j], [\boldsymbol{y}_j])\}_{j\in[1,m]}$ and $[\boldsymbol{z}]$, where $\ell = m/2$.
  2. $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input sharings $([\boldsymbol{a}_{1,1}], \ldots, [\boldsymbol{a}_{1,\ell}])$ and $([\boldsymbol{b}_{1,1}], \ldots, [\boldsymbol{b}_{1,\ell}])$, and then obtains the shares of corrupted verifiers on $[\boldsymbol{c}_1]$.
  3. $\mathcal{S}$ computes the shares of $[\boldsymbol{c}_2] = [\boldsymbol{z}] - [\boldsymbol{c}_1]$ held by corrupted verifiers.
  4. $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ as above for input sharings $([\boldsymbol{a}_{i,1}], \ldots, [\boldsymbol{a}_{i,\ell}])$, $([\boldsymbol{b}_{i,1}], \ldots, [\boldsymbol{b}_{i,\ell}])$ and $[\boldsymbol{c}_i]$ for $i \in [1, 2]$. Then, $\mathcal{S}$ updates the shares of corrupted verifiers on the input packed inner-product tuple $(([\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_{m/2}]), ([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_{m/2}]), [\boldsymbol{z}])$ used in the next iteration, by setting them as the shares of corrupted verifiers on $([\boldsymbol{a}_1], \ldots, [\boldsymbol{a}_\ell])$, $([\boldsymbol{b}_1], \ldots, [\boldsymbol{b}_\ell])$ and $[\boldsymbol{c}]$ output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$.
  5. $\mathcal{S}$ update γ as α output by $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$, and also set $m := m/2$. Then, $\mathcal{S}$ performs the next iteration.

- *Simulation of randomization*: $\mathcal{S}$ holds the shares of $([\boldsymbol{x}_1],[\boldsymbol{x}_2])$, $([\boldsymbol{y}_1],[\boldsymbol{y}_2])$ and $[\boldsymbol{z}]$ held by corrupted verifiers from the dimension-reduction phase. $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:
  1. On behalf of the honest prover, $\mathcal{S}$ samples random elements in $\mathbb{K}$ as the shares of $([\boldsymbol{x}_0],[\boldsymbol{y}_0])$ held by corrupted verifiers.
  2. For $i \in [0,1]$, $\mathcal{S}$ simulates the protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above for input sharings $([\boldsymbol{x}_i],[\boldsymbol{y}_i])$, and then obtains the shares of $[\boldsymbol{z}_i]$ held by corrupted verifiers.
  3. $\mathcal{S}$ computes the shares of corrupted verifiers on $[\boldsymbol{z}_2] := [\boldsymbol{z}] - [\boldsymbol{z}_1]$.
  4. $\mathcal{S}$ simulates the execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$ as described above for input sharings $\{([\boldsymbol{x}_i],[\boldsymbol{y}_i],[\boldsymbol{z}_i])\}_{i\in[0,2]}$, and then obtains the shares of corrupted verifiers on $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$.
  5. $\mathcal{S}$ samples $\boldsymbol{a},\boldsymbol{b} \leftarrow \mathbb{K}^k$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then, $\mathcal{S}$ uses the secrets $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ along with the shares of $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$ held by t corrupted verifiers to reconstruct the *whole* sharings $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$. $\mathcal{S}$ uses the shares of honest verifiers to run the Open procedure with $\mathcal{A}$. For each $i \in \mathcal{H}$, if honest verifier $\mathcal{V}_i$ aborts in the Open procedure, then $\mathcal{S}$ sends abort_i to functionality $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$, otherwise $\mathcal{S}$ sends $\mathsf{continue}_i$ to $\mathcal{F}^{\mathsf{pss}}_{\mathsf{verifyprod}}$.

Hybrid argument. Below, we use a series of hybrids to prove that the real-world execution is perfectly indistinguishable from the ideal-world execution.

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

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates each protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$ as described above. In particular, $\mathcal{S}$ samples uniform elements over $\mathbb{K}$ as the shares of corrupted verifiers and a random vector in $\mathbb{K}^k$ as a broadcast message $\boldsymbol{u}$.

In **Hybrid$_0$**, for every protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{inner\text{-}prod}}$, a random vector $\boldsymbol{r} \in \mathbb{K}^k$ is perfectly hidden from the view of $\mathcal{A}$. Thus, $\boldsymbol{u} = \boldsymbol{z} + \boldsymbol{r}$ is uniform in $\mathbb{K}^k$. In **Hybrid$_0$**, it is easy to see that the shares of corrupted verifiers are uniform in $\mathbb{K}$. Therefore, **Hybrid$_1$** has the identical distribution as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates each protocol execution of $\Pi^{\mathsf{pss}}_{\mathsf{compress}}$, and also simulates the dimension-reduction phase, as described above.

It is clear that **Hybrid$_2$** has the identical distribution as **Hybrid$_1$**, where only the shares of corrupted verifiers are computed by $\mathcal{S}$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the randomization phase as described above. In particular, $\mathcal{S}$ samples random elements in $\mathbb{K}$ as the shares of $[\boldsymbol{x}_0]$ and $[\boldsymbol{y}_0]$ held by corrupted verifiers. $\mathcal{S}$ samples $\boldsymbol{a},\boldsymbol{b}$ at random and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then, $\mathcal{S}$ uses $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ along with the shares of $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$ held by t corrupted verifiers to reconstruct the *whole* sharings $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$. $\mathcal{S}$ uses the shares of honest verifiers to run the Open procedure with $\mathcal{A}$. This is the ideal-world execution.

In **Hybrid$_2$**, $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$ are uniformly random over $\mathbb{K}^k$, and are kept secret from the view of $\mathcal{A}$. Thus, $\boldsymbol{a}$ and $\boldsymbol{b}$ stored in the multiplication tuple $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$ are uniform in $\mathbb{K}^k$. For an honest prover, we always have that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Therefore, $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ in **Hybrid$_2$** has the same distribution as that in **Hybrid$_3$**. In **Hybrid$_3$**, the shares of honest verifiers on $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$ are computed from the secrets $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ and the shares of corrupted verifiers. Thus, the shares of $([\boldsymbol{a}],[\boldsymbol{b}],[\boldsymbol{c}])$ held by honest verifiers in **Hybrid$_3$** has the identical distribution as that in **Hybrid$_2$**. Overall, the distributions of **Hybrid$_2$** and **Hybrid$_3$** are identical.

In conclusion, the real-world execution is indistinguishable from the ideal-world execution, except with probability $\leq \frac{4\lceil \log M \rceil + 5Q_2}{2^\lambda - 3}$, which completes the proof. □