VOProof: Efficient zkSNARKs from Vector Oracle Compilers

Yuncong Zhang
Shanghai Jiao Tong University
shjdzhangyuncong@sjtu.edu.cn

Alan Szepieniec
Nervos
alan@nervos.org

Ren Zhang*
Cryptape Co. Ltd. and Nervos
ren@nervos.org

Shi-Feng Sun*
Shanghai Jiao Tong University
shifeng.sun@sjtu.edu.cn

Geng Wang
Shanghai Jiao Tong University
wanggxx@sjtu.edu.cn

Dawu Gu*
Shanghai Jiao Tong University
dwgu@sjtu.edu.cn

ABSTRACT

The design of zkSNARKs is increasingly complicated and requires familiarity with a broad class of cryptographic and algebraic tools. This complexity in zkSNARK design also increases the difficulty in zkSNARK implementation, analysis, and optimization. To address this complexity, we develop a new workflow for designing and implementing zkSNARKs, called VOProof. In VOProof, the designer only needs to construct a *Vector Oracle (VO) protocol* that is intuitive and straightforward to design, and then feeds this protocol to our *VO compiler* to transform it into a fully functional zkSNARK. This new workflow conceals most algebraic and cryptographic operations inside the compiler, so that the designer is no longer required to understand these cumbersome and error prone procedures. Moreover, our compiler can be fine-tuned to compile one VO protocol into multiple zkSNARKs with different tradeoffs.

We apply VOProof to construct three general-purpose zkSNARKs targeting three popular representations of arithmetic circuits: the Rank-1 Constraint System (R1CS), the Hadamard Product Relation (HPR), and the PLONK circuit. These zkSNARKs have shorter and more intuitive descriptions, thus are easier to implement and optimize compared to prior works. To evaluate their performance, we implement a Python framework for describing VO protocols and compiling them into working Rust code of zkSNARKs. Our evaluation shows that the VOProof-based zkSNARKs have competitive performance, especially in proof size and verification time, e.g., both reduced by roughly 50% compared to Marlin (Chiesa et al., EUROCRYPT 2020). These improvements make the VOProof-based zkSNARKs more preferable in blockchain scenarios where the proof size and verification time are critical.

CCS CONCEPTS

• **Security and privacy** → *Information-theoretic techniques*; • **Theory of computation** → **Cryptographic protocols**; **Interactive proof systems**.

*Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '22, November 7–11, 2022, Los Angeles, CA, USA.
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559387

KEYWORDS

Zero-Knowledge; Proof System; SNARK

ACM Reference Format:
Yuncong Zhang, Alan Szepieniec, Ren Zhang, Shi-Feng Sun, Geng Wang, and Dawu Gu. 2022. VOProof: Efficient zkSNARKs from Vector Oracle Compilers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3548606.3559387

1 INTRODUCTION

Zero-knowledge SNARKs (zkSNARKs), first introduced by Bitansky et al. in 2012 [8], allow a prover to generate a short proof π for a computation output $y = F(x, w)$ of an arbitrary function F and public input x, such that a resource-constrained verifier can validate y with at most $O(\text{polylog}(|F|))$ computation and storage costs while learning nothing about the secret input w. Recent years witnessed a surge of zkSNARKs with various properties, e.g., constant verification time [16, 21, 23, 25], universal setup [5, 13, 16, 18, 21, 25, 28], transparent setup [5, 13, 18, 28], and post-quantum security [5, 7, 18]. New designs emerge rapidly with smaller proof generation and verification costs, shorter proofs, and fewer security assumptions. Despite their short history, zkSNARKs have already been deployed in many blockchain-based scenarios, e.g., Zcash [6], the first fully anonymous cryptocurrency, and Aztec [1] and zkSync [3], two projects boosting the scalability and privacy of Ethereum—the cryptocurrency with the second-largest market capitalization.

Albeit more powerful and efficient, new zkSNARK designs are becoming increasingly complicated. Understanding the mechanism of a zkSNARK requires familiarity with a large and ever-increasing set of cryptographic and non-cryptographic techniques [30]. This complexity makes zkSNARK implementations more cumbersome, error prone, and vulnerable to security flaws. To dissect this complexity, Bünz et al. [13] pointed out that all zkSNARK designs can be described as a three-step workflow, where only the third step involves heavy cryptographic techniques. First, the to-be-verified equation $y = F(x, w)$ is transformed into a *constraint system*, i.e., a mathematical equation over a finite field. Second, an information-theoretic *Polynomial IOP (PIOP)* is designed to verify that the instance-witness pair (x, w) satisfies this constraint system. This is the most time-consuming step among the three, as it often involves sophisticated algebraic techniques: PLONK [21] and Marlin [16] spend 14 and 12 pages, respectively, to describe this step. Finally, the PIOP is compiled into a zkSNARK by a cryptographic compiler called a *polynomial commitment scheme*. This workflow simplifies the zkSNARK construction by reducing the design of zkSNARKs into

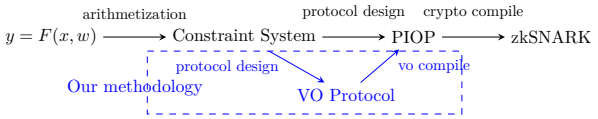


Figure 1: The zkSNARK construction workflow. The VO protocol is more intuitive to design than PIOP, and the VO-to-PIOP compilation is application-agnostic.

that of PIOPs.[1] Observing the bottleneck—the second step—in the workflow, a natural question we ask is:

Can we further simplify the design by concealing the algebraic techniques into another compiler, so that the zkSNARK designers can focus on the application-specific logics?

We address this question by proposing a new zkSNARK design method called VOProof. At the core of this method is a new type of protocol, called *Vector Oracle (VO) protocol*. By the virtue of matching with the constraint systems, whose key components are vector-related operations, VO protocols are considerably more concise than PIOPs: we list three VO protocols for R1CS, HPR and PLONK in two figures, Fig. 2 and Fig. 5, that take only one page in total. After formalizing the notion of VO protocols, we provide a compiler that transforms them into PIOPs without the designer's intervention, replacing the functionalities provided by the vector oracle with algebraic operations. By leveraging the existing cryptographic compilers for the PIOP-to-zkSNARK transformation, our method is illustrated by the new zkSNARK construction pipeline in Fig. 1. In detail, our main contributions are listed as follows:

A Methodology for Simpler zkSNARK Design. We propose a new workflow for designing universal zkSNARKs, named *VOProof* (Sect. 3). In this workflow, the designer first constructs a *VO protocol* introduced in this work, then transforms it into a PIOP via a compiler we propose, and finally compiles the PIOP into a zkSNARK. By wrapping most of the *heavy algebraic and cryptographic operations* in the compilers, VOProof thus conceals these complicated details from the designer and enjoys the two-fold advantage of simplicity and composability. Regarding *simplicity*, the designer or programmer only needs to design or implement a VO protocol, which is highly concise and requires no background in zkSNARKs or cryptography. This also reduces the chance of introducing security flaws in the implementation, as the application-agnostic compiler can be audited and maintained by zkSNARK experts. As for *composability*, we can compile one VO protocol into different zkSNARKs fine-tuning various tradeoffs by configuring the compiler. For example, the zero-knowledge (ZK) property may be turned on/off for different scenarios.

A VO-to-PIOP Compiler. To demonstrate the feasibility of the entire workflow, we present a highly optimized compiler for transforming a VO protocol into a PIOP (Sect. 3.2). Inspired by the techniques in Claymore [29], this compiler uses the canonical monomial basis for encoding vectors into polynomials. We prove that if the VO protocol has completeness and soundness, then the compiled PIOP also has completeness, soundness, and zero-knowledge. VOProof leverages readily available tools for all steps other than VO-to-PIOP compilation; our compiler thus completes the toolchain.

[1]The PIOP formalism also appears in the work of Chiesa et al. [16] by the name *Algebraic Holographic Proof (AHP)*, and Gabizon et al. [21] as *Polynomial Protocols*.

New zkSNARKs for Influential Circuit-based Constraint Systems. We apply VOProof to construct three zkSNARKs targeting three circuit-based constraint systems (Sect. 4), namely VOR1CS for the Rank-1 Constraint System (R1CS), VOHPR for the Hadamard-Product Relation (HPR) and VOPLONK for the constraint system derived from PLONK. These zkSNARKs have more concise descriptions, thus are easier to understand and implement, and have competitive performance compared to the state of the art, as demonstrated by our comprehensive evaluation in Sect. 5.

Implementation and Comprehensive Evaluation. To evaluate the performance of VOProof-based zkSNARKs, we implement a Python framework for describing VO protocols and compiling them into working Rust code of zkSNARKs, based on the VOProof workflow. We then apply this framework to generate the code for the above three zkSNARKs. We benchmark VOR1CS and VOPLONK[2] over the Poseidon-based [22] Merkle path verification and compare the result with three state-of-the-art zkSNARKs Marlin [16], PLONK [21], and Groth16 [23]. The evaluation results demonstrate that the VOProof-based zkSNARKs outperform Marlin and PLONK in both proof sizes and verification times (reduced by roughly 50%). See Table 2 and Fig. 6 (in Sect. 5) for more details.

1.1 Technical Overview

We start with an overview of how VO protocols work and how to transform them into PIOPs, followed by a high-level explanation of why VO protocols are more intuitive and concise than PIOPs.

The Vector Oracle Protocol. A VO protocol involves two parties, the prover and the verifier, where the prover tries to convince the verifier of a statement. Unlike a standard interactive protocol, the parties in a VO protocol have access to an oracle called the vector oracle. The parties may query the oracle to (1) submit arbitrary vectors to the oracle; (2) manipulate submitted vectors by linear combination or right-shifting; (3) check if some vectors satisfy certain equations, e.g., $\boldsymbol{a} \circ \boldsymbol{b} \stackrel{?}{=} \boldsymbol{c} \circ \boldsymbol{d}$, where $\circ$ is the *Hadamard* (entry-wise) product between vectors. After submitting a vector to the oracle, the submitter will receive a *handle*, i.e., a unique identifier, to the vector. The handle contains no information about the vector's content. The parties may send the vector handles to each other. A party can manipulate the vectors not known to itself or check their properties by querying the oracle with the vectors' handles, received either from the other party or the oracle. The manipulation does not modify the existing vectors. Instead, the oracle stores the result in a new vector and returns its handle to the query issuer. Therefore, once a party receives a vector handle, the party is assured that the content of the corresponding vector will never change. In this sense, the vector handle serves as a commitment to the vector.

The two manipulation queries (linear combination and right-shifting) and the property check query (Hadamard equation) are simple yet surprisingly powerful. We demonstrate in Sect. 4 that these functionalities can capture any statements described by arithmetic circuits.

[2]To our knowledge, there are no existing implementations for any HPR-based zkSNARKs, including Sonic [25], BulletProofs [12] (Existing implementations are all R1CS-variant of BulletProofs) and Claymore [29], or circuit composer for HPR. Therefore, it is difficult to make an apples-to-apples comparison between VOHPR and existing HPR-based zkSNARKs for a real-world problem.

VO-to-PIOP Compiler. To compile a VO protocol into a zkSNARK, we first compile the protocol into a PIOP, because the PIOP-to-zkSNARK compilation is extensively studied in the zkSNARK literature [13, 16, 21]. A PIOP is an interactive protocol where the parties have access to *polynomial oracles*. A polynomial oracle plays a similar role as a vector handle in the VO model[3], but serving as the commitment to a polynomial $f(X)$ instead of a vector. The polynomial oracle provides only one functionality to the parties: given any point z, the oracle replies with $f(z)$. In compiling PIOPs to zkSNARKs, the polynomial oracles are replaced by cryptographic polynomial commitments.

Compiling a VO protocol into a PIOP involves four steps. The first step is to establish a one-to-one correspondence between vector handles and polynomial oracles. For every vector $\boldsymbol{v}$ submitted by the prover in the VO protocol, the prover in the compiled PIOP correspondingly sends the oracle of $f_{\boldsymbol{v}}(X) := \sum \boldsymbol{v}_{[i]} X^{i-1}$ to the verifier. For a vector submitted by the verifier, the corresponding polynomial oracle is simulated by the PIOP verifier locally.

The second step is to translate the manipulations to vectors into the manipulations to polynomials. For example, right-shifting a vector $\boldsymbol{v}$ by k positions is translated to multiplying X^k to $f_{\boldsymbol{v}}(X)$. The oracle for the shifted polynomial is simulated by first querying $f_{\boldsymbol{v}}(X)$ at point z, and then multiplying the result with z^k.

Third, the property checks are replaced by PIOP protocols inspired by the techniques in Claymore [29]. We explain the ideas via a concrete example. Suppose the VO verifier queries for checking the Hadamard equation $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{0}$ with vector handles for $\boldsymbol{a}$ and $\boldsymbol{b}$. Recall that the PIOP verifier, via the aforementioned first step, has access to all polynomial oracles corresponding to the vector handles, including the oracles for $f_{\boldsymbol{a}}(X)$ and $f_{\boldsymbol{b}}(X)$. The verifier samples a random ω, and simulates the polynomial oracle for $h(X) := f_{\boldsymbol{a}}(\omega \cdot X^{-1}) \cdot f_{\boldsymbol{b}}(X)$. If the Hadamard equation holds, then the constant term of $h(X)$, denoted by h_0, should be zero. Otherwise, $h_0 = 0$ happens with negligible probability over the randomness of ω, by Schwartz-Zippel Lemma. Therefore, it suffices for the verifier to ensure that $h_0 = 0$. It is difficult for the verifier to check the constant term of a polynomial given only the access to evaluate this polynomial, so the verifier needs help from the prover. To show that the constant term of $h(X)$ is zero, the prover only needs to show that $h(X)$ can be written in the form $\bar{h}(X) - \bar{h}(\gamma \cdot X)$, i.e., the difference between two polynomials whose constant terms are guaranteed to be the same, where γ is a constant. It is easy to find this polynomial $\bar{h}(X)$ as long as the multiplicative order of γ is sufficiently large. We can choose γ to be the generator of the multiplicative group $\mathbb{F}\backslash\{0\}$. The prover computes the appropriate $\bar{h}(X)$ and sends its oracle to the verifier, then the verifier checks the equality $h(X) = \bar{h}(X) - \bar{h}(\gamma \cdot X)$ by querying both sides of the equation at a random point z.

Finally, to ensure that the compiled PIOP is zero-knowledge, the VO-to-PIOP compiler appends randomizer coefficients to every polynomial sent from the prover to the verifier. The randomizers, unfortunately, change the coefficient vector of the polynomials and potentially break the Hadamard equations or other vector properties. Our solution is to modify the property check queries, such as cutting the vectors to keep the first n elements before checking if the vectors satisfy the desired properties. To implement this modified property check, the compiler modifies the Hadamard equations as explained in the following example. If the original equation was $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{0}$, the compiler replaces it with $\boldsymbol{a} \circ \boldsymbol{b} - (\boldsymbol{0}^n \| \boldsymbol{1}) \circ (\boldsymbol{\delta} \| \boldsymbol{t}) = \boldsymbol{0}$ where $\boldsymbol{t}$ is computed and submitted by the prover to satisfy this new equation, and $\boldsymbol{\delta} \in \mathbb{F}^n$ is a random vector. This modification allows the prover to randomize the vectors or polynomials arbitrarily, as long as the modification happens outside the first n elements.

Simplicity of VO Protocols. The advantage of VO protocols in simplicity over PIOPs is best demonstrated via the following example. One common task in protocol design is checking the equality between two groups of elements. In a VO protocol, this problem is usually embodied as comparing parts of two vectors, e.g., to show that the first m elements of vector $\boldsymbol{a}$ equal the first m elements of vector $\boldsymbol{b}$. This task can be accomplished by a single query for checking the Hadamard equation $(\boldsymbol{a} - \boldsymbol{b}) \circ \boldsymbol{1}^m = \boldsymbol{0}$, where $\boldsymbol{1}^m$ is the vector that is filled with ones in the first m positions and zeros elsewhere.

In PIOP, this task often requires transmitting one polynomial oracle in the interaction, and several evaluation queries of the verifier. For example, in Marlin or PLONK, the equality check is embodied as the task of verifying that two polynomials $f(X)$ and $g(X)$ evaluate to the same values over a domain $H \subset \mathbb{F}$. To accomplish this, the prover sends the verifier the polynomial $q(X) := (f(X) - g(X))/Z(X)$ where $Z(X) := \prod_{h \in H}(X - h)$, and the verifier samples a random z and checks the identity $q(z) \cdot Z(z) = f(z) - g(z)$ via three evaluation queries. Describing these operations are already more verbose than a single query in the VO protocol. Moreover, PIOP designers need to manually batch multiple invocations of the above procedure for optimization, which further complicates the protocol description, whereas for the VO protocols, the batching strategy is automatically executed by the compiler.

1.2 Related Works

We classify zkSNARKs into three groups, based on how they achieve *succinctness*—the "S" in zkSNARK. The first group includes BulletProofs [12] and Aurora [7], which achieve logarithmic proof sizes and linear verification complexities. The second group target only *uniform* circuits, i.e., those with very short representations. Examples include Libra [31], which requires the circuit to be layered and *log-space uniform*, and STARK [5] and vRAM [32], which target Random-Access-Machines (RAMs) that are equivalent to circuits consisting of repetitions of the same sub-circuit.

Most zkSNARKs fall into the third group, which introduces preprocessing, allowing the verifier to read a short digest instead of the complete circuit. Spartan [28] and Fractal [18] do not require trusted setups and Fractal is proved post-quantum secure [17]. Pinocchio [26] and Groth16 [23] are pairing-based zkSNARKs that require per-circuit trusted setups. In comparison, Marlin [16], PLONK [21] and Sonic [25] only require a universal trusted setup. These pairing-based zkSNARKs have constant proof sizes and verification complexities.

Supersonic [13] and Claymore [29] cross the group boundaries by focusing on improving the methodology instead of standalone

[3]Unlike vector oracle, a polynomial oracle does not function as a third party, but as objects that can be passed around.

zkSNARKs. Specifically, Supersonic proposes the DARK polynomial commitment and the PIOP formalism, while Claymore explores constructing PIOPs in the monomial-basis setting. Our work is inspired by the techniques proposed in Claymore. Particularly, we share several monomial-basis techniques with Claymore, whereas we generalize their Hadamard protocol and propose an alternative InnerProduct protocol. Other works that improve the modularity of zkSNARK designs include the ILC model [11], the CSS model [27], Lunar [14], and LegoSNARK [15]. Lunar and LegoSNARK focus on Commit-and-Prove SNARKs. Although both ILC and VO operate on vectors, the former focuses on linearity, while VO relies heavily on *non-linear* query—HAD, so the techniques for leveraging and implementing these two models are completely different. Similar to CSS, VO abstracts away the implementation details; our design differs from CSS in our finer levels of granularity.

2 PRELIMINARIES

2.1 Notations

Let $\mathbb{Z}$ be the set of integers. We abbreviate the set $\{i\}_{i=1}^{n}$ by $[n]$, and $\{i\}_{i=m}^{n}$ by $[m..n]$. Throughout this paper, we use a unique finite field $\mathbb{F}$. Denote by $\mathbb{F}^*$ the multiplicative group $\mathbb{F}\backslash\{0\}$. For any set S, denote the size of S by $|S|$.

We denote the vectors by bold lowercase letters. For example, $\boldsymbol{a} \in \mathbb{F}^n$ is a vector of size n over $\mathbb{F}$. We use $\boldsymbol{a}_{[i]}$ to denote the i-th element of $\boldsymbol{a}$. The index starts from one. Let $\boldsymbol{a}_{[i..j]} := (\boldsymbol{a}_{[i]}, \cdots, \boldsymbol{a}_{[j]})$ for $i \le j$. For $i > n$, we treat $\boldsymbol{a}_{[i]} = 0$ for convenience.

We use the following binary operations between vectors. For two vectors $\boldsymbol{a} \in \mathbb{F}^m$ and $\boldsymbol{b} \in \mathbb{F}^n$, their *concatenation* is $\boldsymbol{a}\|\boldsymbol{b} := (\boldsymbol{a}_{[1]}, \cdots, \boldsymbol{a}_{[m]}, \boldsymbol{b}_{[1]}, \cdots, \boldsymbol{b}_{[n]}) \in \mathbb{F}^{m+n}$. Their *sum* is $\boldsymbol{a} + \boldsymbol{b} := (\boldsymbol{a}_{[i]} + \boldsymbol{b}_{[i]})_{i=1}^{\max\{m,n\}} \in \mathbb{F}^{\max\{m,n\}}$. Their *inner product* is $\langle \boldsymbol{a}, \boldsymbol{b} \rangle := \sum_{i \in [\min\{m,n\}]} \boldsymbol{a}_{[i]} \cdot \boldsymbol{b}_{[i]}$. Their *Hadamard (entry-wise) product* is $\boldsymbol{a} \circ \boldsymbol{b} := (\boldsymbol{a}_{[i]} \cdot \boldsymbol{b}_{[i]})_{i=1}^{\min\{m,n\}} \in \mathbb{F}^{\min\{m,n\}}$. We will use *power vectors*, i.e., vectors of the form $(1, \alpha, \alpha^2, \cdots, \alpha^{n-1})$, denoted by $\boldsymbol{\alpha}^n$, where $\alpha \in \mathbb{F}$. In particular, $\boldsymbol{1}^n$ and $\boldsymbol{0}^n$ are the all-one and all-zero vectors of size n. We also use *unit vectors* $\boldsymbol{e}_i$ that has a single one at position i and zeros anywhere else. If we right-shift $\boldsymbol{v} \in \mathbb{F}^n$ by k positions, we get $\boldsymbol{v}^{\rightarrow k} := \boldsymbol{0}^k \| \boldsymbol{v} \in \mathbb{F}^{n+k}$, i.e., prefix the vector with k zeros.

We use bold capital uppercase letters for matrices, e.g., $\boldsymbol{M} \in \mathbb{F}^{m \times n}$ is a matrix of size $m \times n$ over $\mathbb{F}$. $\boldsymbol{M}_{[i,j]}$ is the element of $\boldsymbol{M}$ at the i-th row and j-th column.

We write $f(X) \in \mathbb{F}[X]$ as a polynomial over field $\mathbb{F}$. When the context is clear, we use f_i to represent the coefficient for X^i. For a vector $\boldsymbol{v} \in \mathbb{F}^d$, let $f_{\boldsymbol{v}}(X)$ be the polynomial that uses the elements of $\boldsymbol{v}$ as coefficients, i.e., $f_{\boldsymbol{v}}(X) = \sum_{i=1}^{d} \boldsymbol{v}_{[i]} X^{i-1}$. We call $f_{\boldsymbol{v}}(X)$ the polynomial representation of $\boldsymbol{v}$ in the *canonical monomial basis*.

We write $q(X_1, \cdots, X_\mu) \in \mathbb{F}[X_1, \cdots, X_\mu]^{\le 2}$ as a μ-variate quadratic polynomial. For any $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu \in \mathbb{F}^n$, we write

$$q^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu) := \left(q(\boldsymbol{v}_{1[i]}, \cdots, \boldsymbol{v}_{\mu[i]})\right)_{i=1}^{n} \in \mathbb{F}^n,$$

i.e., evaluating the polynomial q on these vectors using Hadamard product for multiplication. If $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu$ have different sizes, the length of $q^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu)$ is the maximal size of these vectors. Similarly,

$$q^{\langle\cdot,\cdot\rangle}(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu) := \sum_{i=1}^{n} q(\boldsymbol{v}_{1[i]}, \cdots, \boldsymbol{v}_{\mu[i]}) - (n-1)c \in \mathbb{F},$$

is the result of evaluating q over these vectors using inner product for multiplication, where c is the constant term of q, the degree-two terms $X_i X_j$ are evaluated to $\langle \boldsymbol{v}_i, \boldsymbol{v}_j \rangle$, the linear terms X_i are evaluated to $\langle \boldsymbol{v}_i, \boldsymbol{1}^n \rangle$, and subtracting $(n-1)c$ ensures that the constant term is added only once. Obviously, $q^{\langle\cdot,\cdot\rangle}(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu) = \langle q^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu), \boldsymbol{1}^n \rangle - (n-1)c$.

Finally, we introduce the indexed relation, a convenient notion for defining protocols that have offline preprocessings. An indexed relation $\mathcal{R}$ is a set of triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ where $\mathbb{i}$ is the index, $\mathbb{x}$ is the instance, and $\mathbb{w}$ is the witness. The indexed language induced by $\mathcal{R}$ is $\mathcal{L}(\mathcal{R}) := \{(\mathbb{i}, \mathbb{x}) : \exists \mathbb{w}, (\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}\}$.

2.2 Interactive Proof for Indexed Relations

This work focuses on protocols that admit a preprocessing procedure which, on inputting an index of the relation, produces helpful information for the prover and the verifier in the protocol. For convenience, whenever we mention "interactive proof", we are referring to "preprocessing interactive proof", unless otherwise stated. Similarly, by "PIOP" and "zkSNARK", we refer to the preprocessing versions by default.

Definition 2.1 (Preprocessing Interactive Proof). A *preprocessing interactive proof* is a tuple of PPT algorithms $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ named the indexer, the prover, and the verifier, respectively:

- I takes input $\mathbb{i}$ and outputs helpful information $\mathbb{i}_P$ and $\mathbb{i}_V$ for I and V, respectively;
- P takes inputs $\mathbb{i}_P, \mathbb{x}, \mathbb{w}$ and V takes inputs $\mathbb{i}_V, \mathbb{x}$; they interact with each other; in the end, V outputs $b \in \{0, 1\}$.

The above procedure is denoted by $b \leftarrow \langle \mathsf{I}(\mathbb{i}), \mathsf{P}(\mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{x}) \rangle$.

In general, an interactive proof should have *completeness* and *soundness* defined as below.

Completeness: for any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$,

$$\Pr[b = 0 \mid b \leftarrow \langle \mathsf{I}(\mathbb{i}), \mathsf{P}(\mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{x}) \rangle] \le \varepsilon_c,$$

where $\varepsilon_c \in [0, 1/3]$ is called the *completeness error*.

Soundness: for any $(\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R})$ and unbounded algorithm P^*,

$$\Pr[b = 1 \mid b \leftarrow \langle \mathsf{I}(\mathbb{i}), \mathsf{P}^*, \mathsf{V}(\mathbb{x}) \rangle] \le \varepsilon_s.$$

where $\varepsilon_s \in [0, 1/3]$ is called the *soundness error*.

In addition, we say $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ has *perfect completeness* (resp. *soundness*) if $\varepsilon_c = 0$ (resp. $\varepsilon_s = 0$). The protocol is *public-coin* if all the randomnesses used by V are public coins, i.e., they are sent to the prover immediately after they are read by the verifier from the random tape.

In this paper, we focus on *zero-knowledge (ZK)* protocols which require that any (potentially malicious) verifier cannot acquire any information by interacting with the honest prover. This notion is formalized by a simulator that samples the *transcript* indistinguishably from those of honest executions. We recall that the transcript of an execution of $(\mathsf{I}, \mathsf{P}, \mathsf{V})$, denoted by $\mathsf{tr}\langle \mathsf{I}(\mathbb{i}), \mathsf{P}(\mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{x}) \rangle$, consists of $\mathbb{i}_V$ and all the messages exchanged between P and V. In the

context of public-coin protocols, it suffices to consider a weaker version of ZK, called *honest-verifier zero-knowledge (HVZK)*, which only requires that the simulator exists for the honest verifier. Public-coin HVZK proofs can be transformed into zkSNARKs in the random oracle model via the Fiat-Shamir heuristic [19].

Definition 2.2 (Honest-Verifier Zero-Knowledge). The preprocessing interactive proof $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ for the indexed relation $\mathcal{R}$ is honest-verifier zero-knowledge (HVZK) if there exists a PPT algorithm S such that for any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the statistical distance between the distributions of the following two random variables tr and tr' is $O(2^{-(|\mathbb{i}|+|\mathbb{x}|+|\mathbb{w}|)})$:

$$\mathsf{tr} \leftarrow \mathsf{tr}\langle \mathsf{I}(\mathbb{i}), \mathsf{P}(\mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{x})\rangle \quad \text{and} \quad \mathsf{tr}' \leftarrow \mathsf{S}(\mathbb{i}, \mathbb{x}),$$

where the distributions are over all random coins. If the statistical distance is constantly zero, we say this protocol has *perfect ZK*. Otherwise, we say the ZK is *statistical*, or the protocol is HVSZK.

2.3 Polynomial IOP

A *PIOP (Polynomial Interactive Oracle Proof)* [13] is an interactive proof where, unlike in ordinary interactive proofs, the prover and the indexer may send *polynomial oracles* to the verifier. A polynomial oracle encapsulates a polynomial, say $f(X) \in \mathbb{F}[X]$, and replies $f(z)$ when queried with any $z \in \mathbb{F}$. We denote the polynomial oracle for $f(X)$ by $[f(X)]$.

Definition 2.3 (Preprocessing PIOP). A preprocessing PIOP with degree bound D is a public-coin interactive proof $(\mathsf{I}, \mathsf{P}, \mathsf{V})$, except:

- I and P do not send messages to V; instead, they output polynomials of degree less than D.
- For every polynomial $f(X)$ output by I or P, V has access to the *evaluation oracle* of $f(X)$, denoted by $[f(X)]$. Specifically, the verifier may query $[f(X)]$ with arbitrary $z \in \mathbb{F}$ and receives the reply $y = f(z)$.

For PIOP, the notions *completeness* and *soundness* follow directly from those of interactive proofs. However, the definition of HVZK should be handled carefully, as the verifier no longer reads the entire polynomials but only their evaluations at a few points. Moreover, the randomness for computing the evaluation queries should also be simulated, although they do not appear in the verifier messages. Therefore, we define HVSZK for PIOP to be the same as Def. 2.2 except that the notion *transcript* is replaced by a more general notion called the *verifier's view*. Specifically, this view for PIOP consists of (1) $\mathbb{i}_V$, (2) all the verifier randomnesses, and (3) the replies from the polynomial oracles. For an interactive proof, the verifier's view is simply the transcript.

2.4 Universal zkSNARK

A SNARK is a non-interactive protocol with verification and communication costs logarithmic in the statement size. It is known that zkSNARKs exist only in non-plain (e.g., common-reference-string) models [9].

Definition 2.4 (Universal Preprocessing zkSNARK). Let $\mathcal{R}$ be an indexed relation. A zkSNARK for $\mathcal{R}$ is a tuple of four PPT algorithms $(\mathsf{G}, \mathsf{I}, \mathsf{P}, \mathsf{V})$ named the setup, the indexer, the prover and the verifier, respectively:

- G takes the security parameter 1^λ and a size bound D, and outputs a *common reference string* crs;
- I takes crs, an index $\mathbb{i}$ with $|\mathbb{i}| \le D$, and outputs two strings σ_P and σ_V, called the proving key and the verification key respectively[4];
- P takes σ_P and an instance-witness pair $(\mathbb{x}, \mathbb{w})$, and outputs a string π called the proof;
- V takes σ_V, an instance $\mathbb{x}$ and a proof π, and outputs $b \in \{0, 1\}$ indicating whether to accept or not.

A zkSNARK satisfies the following properties:

Completeness: for every D, every $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$ such that $|\mathbb{i}| \le D$, and security parameter λ, $\Pr[b = 1 \mid \mathsf{crs} \leftarrow \mathsf{G}(1^\lambda, D), (\sigma_P, \sigma_V) \leftarrow \mathsf{I}(\mathsf{crs}, \mathbb{i}), \pi \leftarrow \mathsf{P}(\sigma_P, \mathbb{x}, \mathbb{w}), b \leftarrow \mathsf{V}(\sigma_V, \mathbb{x}, \pi)] = 1$.

Argument-of-Knowledge: for any PPT adversary P^*, there exists a PPT extractor E such that for every large enough security parameter λ, every D, and every auxiliary tape z, $\Pr[(\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} \wedge b = 1 \mid \mathsf{crs} \leftarrow \mathsf{G}(1^\lambda, D), (\mathbb{i}, \mathbb{x}, \pi, \sigma_V) \leftarrow \mathsf{P}^*(\mathsf{crs}, z), \mathbb{w} \leftarrow \mathsf{E}(\mathsf{crs}, z), b \leftarrow \mathsf{V}(\sigma_V, \mathbb{x}, \pi)] = O(2^{-\lambda})$.

Succinctness: The proof size is $O(\lambda \cdot \mathrm{polylog}(|\mathbb{i}| + |\mathbb{x}| + |\mathbb{w}|))$ and the running time of V is $O(\lambda \cdot (|\mathbb{x}| + \mathrm{polylog}(|\mathbb{i}| + |\mathbb{w}|)))$.

Zero-knowledge: There exists a PPT simulator S such that for any D, any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$ and security parameter λ, the following two distributions are distinguished with probability $O(2^{-\lambda})$:

$$\left\{ (\sigma_V, \pi) \;\middle|\; \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{G}(1^\lambda, D), \\ (\sigma_P, \sigma_V) \leftarrow \mathsf{I}(\mathsf{crs}, \mathbb{i}), \\ \pi \leftarrow \mathsf{P}(\sigma_P, \mathbb{x}, \mathbb{w}), \\ b \leftarrow \mathsf{V}(\sigma_V, \mathbb{x}, \pi) \end{array} \right\} \text{ and } \left\{ \mathsf{S}(\mathbb{i}, \mathbb{x}, 1^\lambda, D) \right\}.$$

We note that there are zkSNARKs that are non-universal, e.g., Groth16 [23]. The definition of non-universal zkSNARK is the same as Definition 2.4, except that the algorithm G is removed.

Next, we recall a workflow for constructing zkSNARKs which, as pointed out by Bünz et al. [13], is explicitly or implicitly followed by most zkSNARK constructions in the literature.

2.5 The zkSNARK Construction Workflow

Here we illustrate the three-step zkSNARK construction workflow (Fig. 1) in more detail. The starting point of this workflow is a computation, usually represented by an *arithmetic circuit* C that computes a function over $\mathbb{F}$.

The first step of this workflow, called *arithmetization*, transforms the arithmetic circuit into a *constraint system* formalized as indexed relations. Several constraint systems are available for this step. In this paper, we build zkSNARKs for the following popular constraint systems:

- R1CS (Rank-1 Constraint System) is used in Pinocchio [26], Groth16 [23], Aurora [7], Fractal [18], Spartan [28], and Marlin [16].
- HPR (Hadamard Product Relation) was proposed by Bootle et al. [10]. Its variations are used in BulletProofs [12], Sonic [25], and Claymore [29]. A recent lattice-based zkSNARK [4] also chooses this relation.

[4] Also refered to as the circuit-specific CRS.

- The constraint system of PLONK is used solely in PLONK [21], one of the most efficient and popular zkSNARKs.

The second step in the workflow produces a PIOP for verifying the constraint system. This step is where most zkSNARK constructions vary and is the focus of this work.

In the final step, the PIOP is compiled into a zkSNARK via two cryptographic tools. First, replace the polynomial oracles with polynomial commitments and the evaluation queries with an execution of the opening protocol of this polynomial commitment scheme. This step turns the PIOP into an interactive proof in the standard model. Next, this interactive proof is turned into a zkSNARK by the Fiat-Shamir transformation [19] that replaces the verifier messages by query replies from a random oracle approximated by a collision-resistant hash function.

3 THE VOPROOF METHOD

Now we introduce VOProof, a new method for constructing zkSNARKs. This method is inspired by observing a mismatch between the constraint systems, whose key components are vector-related operations, and the PIOPs, in which the main objects are polynomials. Based on this observation, in VOProof, we propose a new type of protocol, called the *Vector Oracle (VO) protocol*, where the parties have access to a rich set of functionalities for operating vectors.

After formalizing the VO protocols, the remaining task is to transform them into zkSNARKs. Since the PIOP-to-zkSNARK compiler is available and extensively studied in the literature [13, 16, 21, 24], it suffices for us to provide a compiler to transform VO protocols into PIOPs.

Next, we illustrate the key components in VOProof, the VO protocols and the VO-to-PIOP compiler, in more detail.

3.1 Vector Oracle Protocols

Vector Oracle Model. To formalize the concept of VO protocols, we propose the *VO model*, in which the protocol participants have access to a *vector oracle* denoted by $\mathcal{O}$. The oracle provides a rich set of vector-related functionalities to the parties. The main purpose of these functionalities is to allow the verifier to check if one or more vectors satisfy certain properties, without reading the content of these (possibly) large vectors. These functionalities can be roughly grouped into three categories: (1) to submit new vectors to the vector oracle; (2) to manipulate existing vectors stored in the oracle; and (3) to check certain properties among the vectors. The details are specified in the following definition.

Definition 3.1 (Vector Oracle). A vector oracle of size n internally maintains a set of vectors that is initially empty. The oracle accepts the following queries:

- $\mathsf{VEC}(\boldsymbol{v} \in \mathbb{F}^m)$: stores $\boldsymbol{v}$ and returns a handle $h_{\boldsymbol{v}}$ to $\boldsymbol{v}$;
- $\mathsf{POW}(\alpha \in \mathbb{F}, k \in [n])$: stores $\boldsymbol{v} := \boldsymbol{\alpha}^k$ and returns $h_{\boldsymbol{v}}$;
- $\mathsf{LIN}(h_{\boldsymbol{v}_1}, \cdots, h_{\boldsymbol{v}_m}, \boldsymbol{c} \in \mathbb{F}^m)$: stores $\boldsymbol{v} := \sum_{i=1}^m \boldsymbol{c}_{[i]} \cdot \boldsymbol{v}_i$ and returns $h_{\boldsymbol{v}}$;
- $\mathsf{SHR}(h_{\boldsymbol{u}}, k \in [n])$: stores $\boldsymbol{v} := \boldsymbol{u}^{\rightarrow k}$ and returns $h_{\boldsymbol{v}}$;
- $\mathsf{HAD}(h_{\boldsymbol{v}_1}, \cdots, h_{\boldsymbol{v}_\mu}, q \in \mathbb{F}[X_1, \cdots, X_\mu]^{\leq 2})$: returns 1 if $q^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu)_{[1..n]} \neq \boldsymbol{0}$, otherwise $\perp$.
- $\mathsf{INN}(h_{\boldsymbol{v}_1}, \cdots, h_{\boldsymbol{v}_\mu}, q \in \mathbb{F}[X_1, \cdots, X_\mu]^{\leq 2})$: returns 1 if $q^{\langle\cdot,\cdot\rangle}(\boldsymbol{v}_{1[1..n]}, \cdots, \boldsymbol{v}_{\mu[1..n]}) \neq 0$, otherwise $\perp$.

The handles contain no information about the vectors. If a query is not of the specified format, or any handle in the query refers to a non-existent vector, the oracle also returns $\perp$.

In practice, μ is a small constant fixed by the protocol. In this work, most HAD queries are used to check equations of the form $\boldsymbol{a} \circ \boldsymbol{b} \stackrel{?}{=} \boldsymbol{c} \circ \boldsymbol{d}$ where $\mu \leq 4$, and likewise the INN queries.

Vector Oracle Protocol. A VO protocol is an interactive protocol where the parties have access to the vector oracle.

Definition 3.2 (Preprocessing VO Protocol). A preprocessing VO protocol with vector size n for an indexed relation $\mathcal{R}$ is a tuple of PPT algorithms $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ that have access to a vector oracle $\mathcal{O}$ with vector size n, such that:

- $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ is a preprocessing interactive protocol for $\mathcal{R}$;
- P does not send any messages to V except vector handles;
- V outputs 1 if and only if the oracle never returns $\perp$.

The notions *completeness*, *soundness*, and *public-coin* follow from preprocessing interactive proofs.

Note that in an honest execution of a VO protocol, the verifier does not receive any information from the prover or $\mathcal{O}$ except the vector handles. Therefore, the view of the verifier simply consists of its own random coins and the vector handles, so any VO protocol trivially satisfies HVZK. We remark that although the VEC query allows the vector to be of arbitrary length, the elements at positions beyond $[1..n]$ will never affect the property checks, so submitting vectors larger than n is unnecessary. We also remark that the verifier should not query VEC with large and dense vectors because we expect the verifier to be efficient in terms of computation time and storage. However, the verifier may use the POW query because this query only requires specifying α and k and the verifier does not need to explicitly store the power vector in the memory. After compiled to PIOP, the corresponding polynomial oracles can be efficiently simulated using the geometric sum.

VO Protocols for Simple Example Relations (Fig. 2). For the readers to better understand how to use the VO model to design protocols, we present VO protocols for several simple relations, mainly for demonstrative purpose. Some of these VO protocols will be used as building blocks for constructing general-purpose zkSNARKs. Specifically, we construct VO protocols for range proof[5] (Equation 1), set commitment with membership proof (Equation 2), vector commitment with batched opening (Equation 3), sum check (Equation 4), product check (Equation 5), permutation check (Equation 6), inner-product check (Equation 7), and subset-sum (Equation 8). We present these VO protocols in Fig. 2.

$$\mathcal{R}_{\mathsf{RP}} := \left\{ \begin{pmatrix} \mathbb{i}: & (n, \ell) \\ \mathbb{x}: & (i, h_{\boldsymbol{v}}) \\ \mathbb{w}: & \boldsymbol{v} \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ i \in [n] \\ \boldsymbol{v}_{[i]} \in [0..2^\ell - 1] \end{array} \right\} \quad (1)$$

$$\mathcal{R}_{\mathsf{Set}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & \left(h_{\boldsymbol{v}}, \{v_i\}_{i=1}^m\right) \\ \mathbb{w}: & \boldsymbol{v} \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ v_i \in \boldsymbol{v}, \forall i \in [m] \\ v_i \neq v_j \text{ for } i \neq j \end{array} \right\} \quad (2)$$

[5]The finite field $\mathbb{F}$ should have large characteristic for range proof to be meaningful. Multiple-entry range proofs can be implemented using techniques from Plookup [20] and are left to future work.

$$\mathcal{R}_{\mathsf{VB}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & \left(h_{\boldsymbol{v}}, (k_i, v_i)_{i=1}^m\right) \\ \mathbb{w}: & \boldsymbol{v} \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{v}} \text{ is a handle} \\ \text{ to } \boldsymbol{v} \in \mathbb{F}^n \\ \boldsymbol{v}_{[k_i]} = v_i, \forall i \in [m] \end{array} \right\} \quad (3)$$

$$\mathcal{R}_{\mathsf{Sum}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & (h_{\boldsymbol{v}}, \ell, c) \\ \mathbb{w}: & \boldsymbol{v} \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ \sum_{i=1}^{\ell} \boldsymbol{v}_{[i]} = c \end{array} \right\} \quad (4)$$

$$\mathcal{R}_{\mathsf{Prod}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & (h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell) \\ \mathbb{w}: & (\boldsymbol{u}, \boldsymbol{v}) \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{u}} \text{ is a handle to } \boldsymbol{u} \in \mathbb{F}^n \\ h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ \prod_{i=1}^{\ell} \boldsymbol{u}_{[i]} = \prod_{i=1}^{\ell} \boldsymbol{v}_{[i]} \end{array} \right\} \quad (5)$$

$$\mathcal{R}_{\mathsf{Perm}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & (h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell) \\ \mathbb{w}: & (\boldsymbol{u}, \boldsymbol{v}) \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{u}} \text{ is a handle to } \boldsymbol{u} \in \mathbb{F}^n \\ h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ \boldsymbol{u}_{[1..\ell]} \text{ and } \boldsymbol{v}_{[1..\ell]} \text{ are} \\ \text{ reorders of each other} \end{array} \right\} \quad (6)$$

$$\mathcal{R}_{\mathsf{IP}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & (h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell, c) \\ \mathbb{w}: & (\boldsymbol{u}, \boldsymbol{v}) \end{pmatrix} \middle| \begin{array}{l} h_{\boldsymbol{u}} \text{ is a handle to } \boldsymbol{u} \in \mathbb{F}^n \\ h_{\boldsymbol{v}} \text{ is a handle to } \boldsymbol{v} \in \mathbb{F}^n \\ \langle \boldsymbol{u}_{[1..\ell]}, \boldsymbol{v}_{[1..\ell]} \rangle = c \end{array} \right\} \quad (7)$$

$$\mathcal{R}_{\mathsf{SubsetSum}} := \left\{ \begin{pmatrix} \mathbb{i}: & n \\ \mathbb{x}: & (\boldsymbol{v}, t) \\ \mathbb{w}: & \boldsymbol{s} \end{pmatrix} \middle| \begin{array}{l} \boldsymbol{s} \in \{0,1\}^n \\ \boldsymbol{v} \in \mathbb{F}^n \\ \langle \boldsymbol{v}, \boldsymbol{s} \rangle = t \end{array} \right\} \quad (8)$$

Note that relations (1) to (7) would be trivial if the witness vectors have not been committed. This is why these relations have vector handles in their instances. If we compile the VO protocols for these relations using a VO-to-SNARK compiler, the vector handles will be replaced with vector commitments, i.e., short random-like strings bound to the vectors, and the VO protocols are transformed into Commit-and-Prove SNARKs [14, 15]. For the subset-sum relation, which is NP-hard, committing the witness vector is unnecessary. Compiling the VO protocol SubsetSum produces a specialized zk-SNARK for the subset-sum problem.

For ease of understanding, we will present the VO protocols using the following descriptions:

- We say "submits $\boldsymbol{v}$" instead of "querying $\mathsf{VEC}(\boldsymbol{v})$", and we will use the name $\boldsymbol{v}$ in the place of its handle $h_{\boldsymbol{v}}$.
- We say "submits $\boldsymbol{\alpha}^k$" instead of "querying $\mathsf{POW}(\alpha, k)$", and we will refer to this vector by $\boldsymbol{\alpha}^k$ instead of its handle. When we need to explicity refer to the handle, we will denote the handle by $h_{\boldsymbol{\alpha}^k}$; similarly for the LIN and SHR queries.
- To describe a HAD query, e.g., $\mathsf{HAD}(h_{\boldsymbol{a}}, h_{\boldsymbol{b}}, h_{\boldsymbol{c}}, q)$, where $q = X_1X_2 - 2X_3 - 4$, we say "checks $\boldsymbol{a} \circ \boldsymbol{b} \stackrel{?}{=}_n 2\boldsymbol{c} + 4 \cdot \boldsymbol{1}^n$", where "$\stackrel{?}{=}_n$" refers to comparing the first n entries between two vectors; similarly for the INN queries.
- To keep the protocol concise, we allow a HAD (or INN) query to refer to a vector submitted by a POW (or LIN, SHR) query without explicitly including their submissions in the description. In this case, we assume this vector has already been submitted by the verifier previously.
- A HAD (or INN) query may use vector concatenation $\boldsymbol{u} \| \boldsymbol{v}$ instead of $\boldsymbol{u} + \boldsymbol{v}^{\rightarrow |\boldsymbol{u}|}$. It may also use sparse vectors, e.g., $\boldsymbol{e}_1 + 2\boldsymbol{e}_3$, assuming they are implicitly submitted by the verifier.

We have introduced the VO protocols and demonstrated how to construct them. Next, we construct the second component of the VOProof method, a compiler that transforms VO protocols into zero-knowledge PIOPs.

3.2 VO-to-PIOP Compiler

Our compiler is based on the canonical monomial basis representation of vectors, i.e., the vectors are embedded into the coefficients of the polynomials. This compiler is partially inspired by the batched Hadamard protocol in Claymore [29]. Here we present a high-level overview of the compiler.

Given any VO protocol (VO.I, VO.P, VO.V), the compiled PIOP, namely (I, P, V), works exactly the same as the VO protocol, except:

1. Vector handles are replaced by polynomial oracles. Specifically, the vector handle $h_{\boldsymbol{v}}$ is replaced by the polynomial oracle $[f_{\boldsymbol{v}}(X)]$. According to the source of this handle, the polynomial oracle is generated in different ways:

- If $h_{\boldsymbol{v}}$ is sent from VO.P (or VO.I) to VO.V, then in the PIOP, P or I sends the polynomial oracle $[f_{\boldsymbol{v}}(X)]$ to V, regardless of the query type from which this handle is returned.
- If $h_{\boldsymbol{v}}$ is returned from a query issued by VO.V itself, then VO.V (thus V) has all the information of the vector $\boldsymbol{v}$ and V may simulate the polynomial oracle $[f_{\boldsymbol{v}}(X)]$ locally, i.e., given any $z \in \mathbb{F}$, V may compute $y = f_{\boldsymbol{v}}(z)$ by itself. Specifically, if $h_{\boldsymbol{v}}$ is returned from a $\mathsf{VEC}(\boldsymbol{v})$ query, V evaluates this polynomial directly using the coefficient vector $\boldsymbol{v}$; for a $\mathsf{POW}(\alpha, k)$ query, V simulates this polynomial oracle by locally computing $\frac{(\alpha \cdot z)^k - 1}{\alpha \cdot z - 1}$ for $\alpha z \neq 1$ or k for $\alpha z = 1$; for a LIN query, V queries the polynomial oracles for the input vectors one by one and linearly combines the results; for a $\mathsf{SHR}(h_{\boldsymbol{v}}, k)$ query, V queries the polynomial oracle $[f_{\boldsymbol{v}}(X)]$ and multiplies the result by z^k.

2. Property checking queries are replaced by interactions. The INN and HAD queries are replaced by interactions between P and V following specific protocols. First, without loss of generality, VO.V may postpone all property check queries to the end of the protocol, such that the INN queries are ordered before the HAD queries. Next, we compile these property check queries into PIOP by the following two steps:

- Replace all the INN queries by an execution of the BatchIP protocol (which is a VO protocol) in Fig. 3. Note that the BatchIP protocol has no INN query. This step removes the INN queries while introduces more vector handles and HAD queries. These vector handles are replaced by polynomial oracles as above, and the HAD queries are processed in the next step together with the remaining HAD queries.
- Replace all the HAD queries by an execution of the BatchHP protocol in Fig. 3. Note that although the input vector list in each INN or HAD query may be different, we can take the union of all these vectors and reindex the variables of the corresponding quadratic polynomials properly, so we only need to input one vector list into the batched inner-product and Hadamard-product protocols.

The above PIOP is not zero-knowledge. To achieve zero-knowledge, P appends r uniformly random coefficients to every polynomial

procedure RangeProof($h_{\boldsymbol{v}}; \boldsymbol{v}$)
P computes $\boldsymbol{b} \in \{0,1\}^{\ell} \subset \mathbb{F}^{\ell}$, the bit-expansion of $\boldsymbol{v}_{[i]}$;
P submits $\boldsymbol{b}$ and sends $h_{\boldsymbol{b}}$ to V and V checks $\boldsymbol{b} \circ \boldsymbol{b} \stackrel{?}{=}_n \boldsymbol{b}$;
V checks $\langle \boldsymbol{b}, 2^{\ell} \rangle - \langle \boldsymbol{v}, \boldsymbol{e}_i \rangle \stackrel{?}{=} 0$.

procedure SetMember($h_{\boldsymbol{v}}, \{v_i\}_{i=1}^m; \boldsymbol{v}$)
P reorders $\boldsymbol{v}$ into $\boldsymbol{v}'$ such that $\boldsymbol{v}'_{[i]} = v_i$ for $i \in [m]$, submits $\boldsymbol{v}'$ and sends $h_{\boldsymbol{v}'}$ to V;
V checks $\left(\boldsymbol{v}' - \sum_{i=1}^m v_i \cdot \boldsymbol{e}_i\right) \circ \left(\sum_{i=1}^m \boldsymbol{e}_k\right) \stackrel{?}{=}_n \boldsymbol{0}$;
Run PermCheck($h_{\boldsymbol{v}}, h_{\boldsymbol{v}'}, n; \boldsymbol{v}, \boldsymbol{v}'$).

procedure VBatch($h_{\boldsymbol{v}}, \{k_i, v_i\}_{i=1}^m; \boldsymbol{v}$)
V checks $\left(\boldsymbol{v} - \sum_{i=1}^m v_i \cdot \boldsymbol{e}_{k_i}\right) \circ \left(\sum_{i=1}^m \boldsymbol{e}_{k_i}\right) \stackrel{?}{=}_n \boldsymbol{0}$.

procedure SumCheck($h_{\boldsymbol{v}}, \ell, c; \boldsymbol{v}$)
V checks $\langle \boldsymbol{v} - c \cdot \boldsymbol{e}_1, \boldsymbol{1}^{\ell} \rangle \stackrel{?}{=} 0$.

procedure ProdCheck($h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell; \boldsymbol{u}, \boldsymbol{v}$)
P submits $\boldsymbol{r} := \left(\prod_{j \in [i]} \frac{\boldsymbol{u}_{[j]}}{\boldsymbol{v}_{[j]}}\right)_{i=1}^{\ell}$ and sends $h_{\boldsymbol{r}}$ to V;
V checks $\boldsymbol{r}^{\to n-\ell} \circ \boldsymbol{v}^{\to n-\ell} \stackrel{?}{=}_n (1 \| \boldsymbol{r})^{\to n-\ell} \circ \boldsymbol{u}^{\to n-\ell}$;
V checks $\boldsymbol{r} \circ \boldsymbol{e}_{\ell} \stackrel{?}{=}_n \boldsymbol{e}_{\ell}$.

procedure PermCheck($h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell; \boldsymbol{u}, \boldsymbol{v}$)
V samples $\beta \stackrel{\$}{\leftarrow} \mathbb{F}$ and sends β to P;
Run ProdCheck($h_{\boldsymbol{u}+\beta \cdot \boldsymbol{1}^{\ell}}, h_{\boldsymbol{v}+\beta \cdot \boldsymbol{1}^{\ell}}, \ell; \boldsymbol{u} + \beta \cdot \boldsymbol{1}^{\ell}, \boldsymbol{v} + \beta \cdot \boldsymbol{1}^{\ell}$).

procedure InnerProductCheck($h_{\boldsymbol{u}}, h_{\boldsymbol{v}}, \ell, c; \boldsymbol{u}, \boldsymbol{v}$)
V checks $\langle \boldsymbol{u}^{\to n-\ell}, \boldsymbol{v}^{\to n-\ell} \rangle - c \stackrel{?}{=} 0$.

procedure SubsetSum($\boldsymbol{v}, t; \boldsymbol{s}$)
P submits $\boldsymbol{s}$ and sends $h_{\boldsymbol{s}}$ to V;
V checks $\boldsymbol{s} \circ \boldsymbol{s} \stackrel{?}{=}_n \boldsymbol{s}$ and $\langle \boldsymbol{v}, \boldsymbol{s} \rangle \stackrel{?}{=} t$.

Figure 2: Demonstrative VO Protocols for Simple Relations. The input to each protocol has two parts, separated by semicolon. The first part is the public input learned by both parties, and the second part is the witness learned only by the prover.

sent to V, where the choice of r is discussed in the following proof sketch. These random coefficients are positioned appropriately to avoid breaking the HAD and INN queries.

We summarize the properties of the above compiler in Theorem 3.3.

Theorem 3.3. *Let* (VO.I, VO.P, VO.V) *be a preprocessing VO protocol with vector size n that verifies the indexed relation $\mathcal{R}$ with completeness error ε_c and soundness error ε_s. Assume that* VO.I *submits m vectors,* VO.P *submits t vectors,* VO.V *issues t_H* HAD *queries, t_I* INN *queries. Then there exists a preprocessing PIOP protocol* (I, P, V) *with degree bound at most $4n+1$ that verifies $\mathcal{R}$ that has completeness error ε_c and soundness error $\varepsilon_s + \frac{t_H+t_I+6n+6}{|\mathbb{F}|-2}$, and satisfies HVZK. Moreover,* I *sends m polynomial oracles to* V, P *sends $t+3$ polynomial oracles to* V, *and* V *makes at most $2(m+t)+6$ evaluation queries at 3 distinct points.*

If $t_I = 0$, i.e., the VO protocol does not use any INN *queries,* P *sends $t+2$ online polynomial oracles, and the verifier makes at most $2(m+t)+4$ evaluation queries at 3 distinct points.*

First, we introduce the Schwartz-Zippel lemma, which is used intensively in all the proofs thereafter.

Lemma 3.4 (Schwartz-Zippel). *Let $f(X_1, \cdots, X_u)$ be a u-variate polynomial of total degree d over $\mathbb{F}$, S be a finite subset of $\mathbb{F}$, and $z_1, \cdots, z_u$ be selected at random independently and uniformly from S. Then*

$$\Pr[f(z_1, \cdots, z_u) = 0] \leq \frac{d}{|S|}.$$

Proof of Theorem 3.3. We will define a sequence of models $\{\mathsf{VO}_i\}$ that starts from the VO model and ends with an alternative formalization of the PIOP model. Correspondingly, starting from the original VO protocol (VO.I, VO.P, VO.V), we construct a sequence of protocols $\{(\mathsf{VO.I}^{(i)}, \mathsf{VO.P}^{(i)}, \mathsf{VO.V}^{(i)})\}$, one for each of the models, and the last of the sequence would be the PIOP protocol with the desired properties.

VO_0. This is exactly the VO model.

VO_1. VO_1 is the same as VO_0, except that the vector handle lists in HAD and INN queries include all the vector handles that is ever returned from the oracle, listed in the order of the time they are returned. The quadratic polynomials in the queries do not change, except that the indeterminants are reindexed accordingly. Since all the HAD and INN queries have the same vector handle list, we can simply omit it from the query syntax. This step is merely a change of notation, such that when we can conveniently add the quadratic polynomials in different queries together.

VO_2. VO_2 is the same as VO_1, except that the queries $\mathsf{VEC}(\boldsymbol{v})$ allow $|\boldsymbol{v}| \geq n$. Clearly, this change only affects the part of the vectors outside the $[1..n]$ window, and VO_2 is equivalent to VO_1.

We also modify a bit of the protocol in this model. Specifically, let $\mathsf{VO.P}^{(2)}$ be the same as $\mathsf{VO.P}^{(1)}$ except that for any vector $\boldsymbol{v}$ submitted by $\mathsf{VO.P}^{(1)}$, $\mathsf{VO.P}^{(2)}$ uniformly randomly samples $\boldsymbol{\delta} \in \mathbb{F}^r$ and submits $\boldsymbol{v} \| \boldsymbol{\delta} \in \mathbb{F}^{n+r}$ instead, where r is a small integer to be determined later. This does not affect completeness since the appended $\boldsymbol{\delta}$ never affects the part of the vectors in the window, nor soundness which does not rely on the prover.

VO_3. VO_3 is the same as $\mathsf{VO.P}^{(2)}$, except that the LIN query is removed. We then modify the verifier as follows. $\mathsf{VO.V}^{(3)}$ is the same as $\mathsf{VO.V}^{(2)}$ except that all the LIN queries are not issued, and for every HAD or INN query that involves one or more vector handles returned from LIN queries, those vectors are substituted by the linear combination of vectors as defined by the LIN query. Specifically, if $h_{\boldsymbol{v}}$ is returned from the query $\mathsf{LIN}(h_{\boldsymbol{v}_1}, \cdots, h_{\boldsymbol{v}_k, \boldsymbol{c}})$, then in every HAD or INN query where $h_{\boldsymbol{v}}$ appears, $h_{\boldsymbol{v}}$ is removed from the vector handle list. Assume without loss of generality that $h_{\boldsymbol{v}}$ is the first parameter to the HAD query, then X_1 in the quadratic polynomial is substituted by $\boldsymbol{c}_{[i_1]} X_{i_1} + \cdots + \boldsymbol{c}_{[i_k]} X_{i_k}$, where i_j is the index of $h_{\boldsymbol{v}_j}$ in the new vector handle list. It is straightforward to check that $\mathsf{VO.V}^{(3)}$ is equivalent to $\mathsf{VO.V}^{(2)}$.

VO_4. VO_4 is the same as VO_3, except that the verifier can make at most one INN query. If $\mathsf{VO.V}^{(3)}$ does not issue any INN quries, then $\mathsf{VO.V}^{(4)}$ is the same as $\mathsf{VO.V}^{(3)}$. Otherwise, assume that the INN queries made by $\mathsf{VO.V}^{(3)}$ are $\mathsf{INN}(q_1), \cdots, \mathsf{INN}(q_{t_I})$. Then

procedure BatchIP($\{h_{\boldsymbol{v}_i}\}_{i=1}^{\mu}, \{q_i(X_1, \cdots, X_\mu)\}_{i=1}^{m}; \{\boldsymbol{v}_i\}_{i=1}^{\mu}$)
VO.V samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends β to VO.P;
Run InnerProduct($\{h_{\boldsymbol{v}_i}\}_{i=1}^{\mu}, \sum_{i=1}^{m} \beta^{i-1} q_i; \{\boldsymbol{v}_i\}_{i=1}^{\mu}$).

procedure BatchHP($\{[f_{\boldsymbol{v}_i}(X)]\}_{i=1}^{\mu}, \{q_i(X_1, \cdots, X_\mu)\}_{i=1}^{m}; \{\boldsymbol{v}_i\}_{i=1}^{\mu}$)
V samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends α to P;
Run Hadamard($\{[f_{\boldsymbol{v}_i}(X)]\}_{i=1}^{\mu}, \sum_{i=1}^{m} \alpha^{i-1} q_i; \{\boldsymbol{v}_i\}_{i=1}^{\mu}$).

procedure InnerProduct($\{h_{\boldsymbol{v}_i}\}_{i=1}^{\mu}, q(X_1, \cdots, X_\mu); \{\boldsymbol{v}_i\}_{i=1}^{\mu}$)
Let $q(X_1, \cdots, X_\mu) = q_0(X_1, \cdots, X_\mu) + c$ where q_0 has zero constant and $c \in \mathbb{F}$;
VO.P computes $\boldsymbol{r} := q_0^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu)$;
VO.P submits $\tilde{\boldsymbol{r}} := \left(\sum_{j=1}^{i} \boldsymbol{r}_{[j]}\right)_{i=1}^{n}$ and sends $h_{\tilde{\boldsymbol{r}}}$ to VO.V;
VO.V checks $\tilde{\boldsymbol{r}} - \tilde{\boldsymbol{r}}^{\rightarrow 1} \stackrel{?}{=}_n f_0^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu)$ and $(\tilde{\boldsymbol{r}} + c \cdot \boldsymbol{e}_n) \circ \boldsymbol{e}_n \stackrel{?}{=}_n \boldsymbol{0}$.

procedure Hadamard($\{[f_{\boldsymbol{v}_i}(X)]\}_{i=1}^{\mu}, q(X_1, \cdots, X_\mu); \{\boldsymbol{v}_i\}_{i=1}^{\mu}$)
P samples $\boldsymbol{\delta} \xleftarrow{\$} \mathbb{F}^r$;
P computes $\boldsymbol{t} := \boldsymbol{\delta} \| q^\circ(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_\mu)_{[n..]}$ and sends $[f_{\boldsymbol{t}}(X)]$ to V;
Let $\boldsymbol{v}_{\mu+1} := \boldsymbol{0}^n \| \boldsymbol{1}^n$ and V simulates $[f_{\boldsymbol{v}_{\mu+1}}(X)]$ locally;
Let $\boldsymbol{v}_{\mu+2} := \boldsymbol{t}^{\rightarrow n-r}$ and V simulates $[f_{\boldsymbol{v}_{\mu+2}}(X)]$ by querying $[f_{\boldsymbol{t}}(X)]$ with z and multiplying the with z^{n-r};
Let $p(X_1, \cdots, X_{\mu+2}) := q(X_1, \cdots, X_\mu) - X_{\mu+1} \cdot X_{\mu+2}$;
Assume $p(X_1, \cdots, X_{\mu+1}) = \sum_{i,j} p_{ij} \cdot X_i X_j + \sum_i p_i \cdot X_i + c$;
V samples $\omega \xleftarrow{\$} \mathbb{F}$ and sends ω to P;
P computes $h(X) := \sum_{i,j} p_{ij} \cdot f_{\boldsymbol{v}_i}(\omega X^{-1}) f_{\boldsymbol{v}_j}(X) + \sum_i p_i \cdot f_{\boldsymbol{1}^n}(\omega X^{-1}) f_{\boldsymbol{v}_i}(X) + c \cdot \frac{\omega^n - 1}{\omega - 1}$;
Assume $h(X) = \sum_{1 \le i \le D} h_{-i} X^{-i} + h_i X^i$;
P computes $\bar{h}_i := h_i / (\gamma^i - 1)$ for $i \neq 0$ and samples $\bar{h}_0 \xleftarrow{\$} \mathbb{F}$, where γ is the generator of the multiplicative group $\mathbb{F}^*$;
P computes $\bar{h}(X) := \sum_i \bar{h}_i X^i$ and sends $[X^D \bar{h}(X)]$ to V;
V samples $z \xleftarrow{\$} \mathbb{F}$ and checks the identity $h(z) \stackrel{?}{=} \bar{h}(\gamma z) - \bar{h}(z)$, where $h(X)$ is evaluated by querying $f_{\boldsymbol{v}_i}(X)$ with z or ωz and composing the results according to the definition of $h(X)$, and $\bar{h}(X)$ is evaluated by querying $[X^D \bar{h}(X)]$.

Figure 3: Batched Inner Product and Hadamard Product Protocols.

$\mathsf{VO.V}^{(4)}$ is the same as $\mathsf{VO.V}^{(3)}$ except that these INN queries are removed, and instead $\mathsf{VO.V}^{(4)}$ samples $\beta \xleftarrow{\$} \mathbb{F}$ and queries $\mathcal{O}$ with a single inner-product query $\mathsf{INN}(q_1 + \cdots + \beta^{t_I - 1} \cdot q_{t_I})$. It is easy to check that $f^{\langle \cdot, \cdot \rangle}$ is linear w.r.t. its inputs. By Schwartz-Zippel Lemma, if the original INN queries are accepted, this new INN query will also be accepted. On the other hand, if at least of the original INN queries is rejected, then this new INN query rejects with probability at least $1 - t_I/|\mathbb{F}|$. Therefore, the soundness error of $\mathsf{VO.V}^{(4)}$ increases by $t_I/|\mathbb{F}|$ compared to the last step.

VO_5. VO_5 is the same as VO_4, except that the verifier cannot make any INN queries. If $\mathsf{VO.V}^{(4)}$ does not make INN query then $\mathsf{VO.V}^{(5)}$ is the same as $\mathsf{VO.V}^{(4)}$. Otherwise, $\mathsf{VO.V}^{(5)}$ is the same as $\mathsf{VO.V}^{(4)}$ except that the one INN query $\mathsf{INN}(q)$ is replaced by an invocation of the protocol InnerProduct in Fig. 3.

To justify this change, note that by definition, the sum of elements in $q^\circ(\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(\mu)})_{[1..n]}$ equals $q_0^{\langle \cdot, \cdot \rangle}(\boldsymbol{v}_{1[1..n]}, \cdots, \boldsymbol{v}_{\mu[1..n]})$. When the prover is honest and the original INN query is accepted, i.e., $q_0^{\langle \cdot, \cdot \rangle}(\boldsymbol{v}_{1[1..n]}, \cdots, \boldsymbol{v}_{\mu[1..n]}) + c = 0$, then the sum of elements in $\boldsymbol{r}$ is c. By the definition of $\tilde{\boldsymbol{r}}$, we have $\tilde{\boldsymbol{r}}_{[n]} = c$. It is straightforward to check that $\mathsf{VO.V}^{(5)}$ accepts.

On the other hand, if $\mathsf{VO.V}^{(5)}$ accepts, then we know that there exists a vector $\tilde{\boldsymbol{r}}$ such that $\tilde{\boldsymbol{r}}_{[n]} = c$ and that $\tilde{\boldsymbol{r}}_{[i]} - \tilde{\boldsymbol{r}}_{[i-1]} = 0$ for every $i \in [n]$ (where $\tilde{\boldsymbol{r}}_{[0]}$ is defined as 0) hold simultaneously. By defining $\boldsymbol{r} := \left(\tilde{\boldsymbol{r}}_{[i]} - \tilde{\boldsymbol{r}}_{[i-1]}\right)_{i=1}^{n} = \tilde{\boldsymbol{r}} - \tilde{\boldsymbol{r}}^{\rightarrow 1}$ and the fact that $\mathsf{VO.V}^{(5)}$ accepts, we conclude that the sum of the elements in $q^\circ(\boldsymbol{v}_{1[1..n]}, \cdots, \boldsymbol{v}_{\mu[1..n]})_{[1..n]}$ is $\tilde{\boldsymbol{r}}_{[n]} = c$, which means $\mathsf{VO.V}^{(4)}$ should also accepts the $\mathsf{INN}(q + c)$ query. We conclude that the soundness error of VO_5 is the same as that of VO_4.

VO_6. VO_6 modifies VO_5 by requiring that the verifier issues at most one HAD query. We then modify the verifier to suit this new model as follows. If $\mathsf{VO.V}^{(5)}$ does not make any HAD queries, then $\mathsf{VO.V}^{(6)}$ is the same as $\mathsf{VO.V}^{(5)}$. Otherwise, assume $\mathsf{VO.V}^{(5)}$ makes the HAD queries $\mathsf{HAD}(q_1), \cdots, \mathsf{HAD}(q_{t_H})$. Then $\mathsf{VO.V}^{(6)}$ is the same as $\mathsf{VO.V}^{(5)}$ except that the HAD queries are removed, and $\mathsf{VO.V}^{(6)}$ instead execute the following procedure. $\mathsf{VO.V}^{(6)}$ samples $\alpha \xleftarrow{\$} \mathbb{F}$ and issues a single HAD query $\mathsf{HAD}(q_1 + \cdots + \alpha^{t_H - 1} \cdot q_{t_H})$. By the same argument as for VO_4, the soundness error of $\mathsf{VO.V}^{(6)}$ increases by $t_H/|\mathbb{F}|$ compared to the previous step.

VO_7. VO_7 modifies VO_6 by adding a new query $\mathsf{EVAL}(h_{\boldsymbol{v}}, z)$ which replies with $f_{\boldsymbol{v}}(z)$. This query is only available to the verifier, thus does not affect soundness of the protocol.

VO_8. VO_8 modifies VO_7 by removing the HAD query. We then modify the protocol to replace the only HAD query by the Hadamard protocol in Fig. 3, where the polynomial oracles are viewed as vector handles and queries to the polynomial oracles are replaced by EVAL queries to the vector handle.

We show that the probability of verifier acceptance is negligibly different from that of the original HAD query. Note that the constant term of $h(X)$ is a polynomial of ω of degree at most $D := 2n + r$ and the coefficient vector of this polynomial of ω is exactly $q^\circ(\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(\mu)})$. Therefore, if the original HAD query issued by $\mathsf{VO.V}^{(7)}$ is accepted, then the constant term of $h(X)$ is zero. In this case, $\mathsf{VO.V}^{(8)}$ accepts with probability 1.

On the other hand, if the HAD query by $\mathsf{VO.V}^{(7)}$ is not accepted, then for uniformly random ω, by Schwartz-Zippel Lemma, the constant term of $h(X)$ is nonzero except with probability $(2n + r)/|\mathbb{F}|$. However, for any polynomial $\bar{h}(X)$, the constant term of $\bar{h}(\gamma X)$ must be the same as the constant term of $\bar{h}(X)$, so $\mathsf{VO.P}^{(8)}$ cannot find any $\bar{h}(X)$ such that $h(X) = \bar{h}(\gamma X) - \bar{h}(X)$ hold. Therefore, by Schwartz-Zippel Lemma, for uniformly random z, the equality

$h(z) = \bar{h}(\gamma z) - \bar{h}(z)$ holds with probability $(4n + 2r)/|\mathbb{F}|$. We conclude that the soundness error of $\mathsf{VO.V}^{(8)}$ increases by less than $(6n + 3r)/|\mathbb{F}|$ compared to the last step.

VO_9. VO_9 modifies VO_8 by requiring that for any $\mathsf{EVAL}(h_{\boldsymbol{v}}, z)$ query, the vector handle $h_{\boldsymbol{v}}$ must be returned from a VEC query.

We modify the protocol subjecting to the above restriction as follows. Let $\mathsf{VO.V}^{(9)}$ be the same as $\mathsf{VO.V}^{(8)}$ except that for every $\mathsf{EVAL}(h_{\boldsymbol{v}}, z)$ query, if the $h_{\boldsymbol{v}}$ is not returned from a VEC query, $\mathsf{VO.V}^{(9)}$ instead executes one of the following:

(1) if $h_{\boldsymbol{v}}$ is returned from a POW query, say $\mathsf{POW}(\alpha, k)$, $\mathsf{VO.V}^{(9)}$ computes the reply by $y = \frac{(\alpha z)^k - 1}{\alpha z - 1}$ or $y = k \cdot 1$ if $\alpha z = 1$;
(2) if $h_{\boldsymbol{v}}$ is returned from an SHR query, say $\mathsf{SHR}(h_{\boldsymbol{u}}, k)$, $\mathsf{VO.V}^{(9)}$ first queries for $y_s = \mathsf{EVAL}(h_{\boldsymbol{u}}, z)$, then computes the reply of this query by $y = z^k \cdot y_s$.

The equivalence between $\mathsf{VO.V}^{(9)}$ and $\mathsf{VO.V}^{(8)}$ follows directly from the definition of EVAL query and the POW, SHR queries.

VO_{10}. VO_{10} modifies VO_9 by removing the POW and SHR queries. Since the vectors submitted by these types of queries are never referenced, we can safely remove all these queries from the protocol.

VO_{10} has only two types of queries left: VEC and EVAL. This model is equivalent to the PIOP model, and the transformation from the protocol $(\mathsf{VO.I}^{(10)}, \mathsf{VO.P}^{(10)}, \mathsf{VO.V}^{(10)})$ to a PIOP protocol $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ is straightforward: every VEC query corresponds to a polynomial oracle sent from the indexer or the prover to the verifier, and every EVAL query corresponds to an evaluation query to a polynomial oracle. Note that the degree bound for the protocol is *at least* $4n + r - 1$, which is one more than the maximal degree of the polynomial $\bar{h}(X)$. However, in PIOP model with higher degree bound, the protocol also works, since VO_{10} does not limit the size of the vectors submitted by the prover.

The number of polynomials sent by the indexer is still m, while the number of polynomials sent by the prover is $t + 3$, where the three additional polynomials are $f_{\tilde{\boldsymbol{r}}}(X)$, $f_{\boldsymbol{t}}(X)$, and $\bar{h}(X)$. Note that in the final PIOP, the polynomial $f_{\tilde{\boldsymbol{r}}}(X)$ is queried at most once, $f_{\boldsymbol{t}}(X)$ is queried at most once, and $\bar{h}(X)$ is queried at most twice, i.e., at z and $\gamma \cdot z$ respectively. For every vector submitted by the VEC query in the original protocol, this vector is queried at most twice in the end, once at z and another time at $\omega \cdot z^{-1}$. Therefore, there are at most $2(m + t) + 4$ evaluation queries at 3 distinct points. If the original protocol does not make any INN query, i.e., $t_I = 0$, the vector $\tilde{\boldsymbol{r}}$ is no longer necessary, the prover polynomials becomes $t + 2$ and the number of evaluation queries becomes $2(m + t) + 3$.

The completeness error is ε_c, and the soundness error is bound by $\varepsilon_s + (t_I + t_H + 6n + 3r)/(|\mathbb{F}| - 1)$.

Honest-Verifier Zero-Knowledge. We show that the PIOP protocol $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ is honest-verifier zero-knowledge. Note that every polynomial sent by P contains r fresh uniformly random coefficients $\boldsymbol{\delta} \in \mathbb{F}^r$, except $f_{\boldsymbol{h}}(X)$, contains only one random coefficient $\delta \in \mathbb{F}$.

We construct a simulator S that given $\mathbb{i}, \mathbb{x}$ samples the verifier view. The verifier view contains the following values: the verifier messages, i.e., the verifier messages in the original protocol together with α, β, ω, z, and the responses from the evaluation queries, i.e., $u_i := f_{\boldsymbol{v}_i}(\omega \cdot z^{-1}), v_i := f_{\boldsymbol{v}_i}(z)$ for $i \in [m + t + 2]$ where $\boldsymbol{v}_i$ is the i-th vector submitted by the VEC query, and $y_1 = (\gamma \cdot z)^{2n+r} \bar{h}(\gamma \cdot z)$, $y_2 = z^{2n+r} \bar{h}(z)$.

The simulator S samples the verifier view by simulating a run of the protocol that differs from an honest run in the following respects:

- the prover sends dummy polynomial oracles to the verifier;
- for each evaluation query:
  - if it is a query for $f_{\boldsymbol{v}_i}(\omega z^{-1})$ or $f_{\boldsymbol{v}_i}(z)$ where $i \in [m]$, i.e., this vector is submitted by the indexer, since S has access to the index, S may compute $\boldsymbol{v}_i$ and therefore the polynomial evaluations accordingly;
  - if it is a query for $f_{\boldsymbol{v}_i}(\omega z^{-1})$ or $f_{\boldsymbol{v}_i}(z)$ where $i \in [m+1..m+t+2]$ or for y_2, i.e., this vector is submitted by the prover, S uniformly randomly sample the query result from $\mathbb{F}$;
  - finally, for the query $y_1 = \bar{h}(\gamma \cdot z)$, compute y_1 according to the identity $h(z) = \bar{h}(\gamma \cdot z) - \bar{h}(z)$.

We show that the output of the above-defined S only has a negligible statistical difference from the verifier view. Since S has access to all the information that the verifier has, the verifier messages simulated by S follow exactly the same distribution of that of an honest run of the protocol. We only need to show that the query results u_i, v_i for $i \in [m + 1..m + t + 2]$ and y_2 in the real execution are uniformly random over $\mathbb{F}$ independent of the rest of the verifier view. Consider the following matrices:

$$\boldsymbol{V} = \begin{pmatrix} \boldsymbol{v}_{m+1}^{\mathsf{T}} \\ \boldsymbol{v}_{m+2}^{\mathsf{T}} \\ \vdots \\ \boldsymbol{v}_{m+t+2}^{\mathsf{T}} \end{pmatrix} \quad \boldsymbol{X} = \begin{pmatrix} 1 & 1 \\ \omega \cdot z^{-1} & z \\ (\omega \cdot z^{-1})^2 & z^2 \\ \vdots & \vdots \\ (\omega \cdot z^{-1})^{n+r-1} & z^{n+r-1} \end{pmatrix}.$$

Note that every 2 rows of matrix $\boldsymbol{X}$ form an invertible sub-matrix except when $\omega \cdot z^{-1} = z$ which happens with probability bounded by $\frac{1}{|\mathbb{F}|}$. However, we can avoid this narrow case by letting the verifying (and the simulator) resample z, which increases the soundness error very slightly (the denominator is changed from $|\mathbb{F}| - 1$ to $|\mathbb{F}| - 2$). Also note that every row of matrix $\boldsymbol{V}$ contains r uniformly random elements in $\mathbb{F}$. Let $r = 2$, then we have

$$\begin{pmatrix} f_{\boldsymbol{v}_{m+1}}(\omega z^{-1}) & f_{\boldsymbol{v}_{m+1}}(z) \\ f_{\boldsymbol{v}_{m+2}}(\omega z^{-1}) & f_{\boldsymbol{v}_{m+2}}(z) \\ \vdots & \vdots \\ f_{\boldsymbol{v}_{m+t+2}}(\omega z^{-1}) & f_{\boldsymbol{v}_{m+t+2}}(z) \end{pmatrix} = \boldsymbol{V}\boldsymbol{X}$$

is uniformly random over $\mathbb{F}^{(r+2)\times 2}$. Finally, since $\boldsymbol{h}$ contains one element $\delta \in \mathbb{F}$ that is uniformly random over $\mathbb{F}$, y_2 is also uniformly random. In conclusion, the output of S has exactly the same distribution with the verifier view.

□

We remark that the upper bounds on polynomial oracles and evaluation queries given in Theorem 3.3 are rather loose and are achieved only in extreme cases: in our work, the compiled PIOPs have much smaller numbers. On the other hand, the soundness error in Theorem 3.3 is precise: directly analyzing the resulting PIOPs gives the same soundness errors.

The above VO-to-PIOP compiler uses the monomial basis. This compiler can alternatively be implemented using the Reed-Solomon (RS) code basis, where a vector is identified by its interpolation polynomials over a structured domain $H \subset \mathbb{F}$. The RS code is an important component in many zkSNARKs. To construct RS code based compiler, however, the VO model should be modified to avoid several efficiency issues.

- In the monomial basis, the polynomial for power vector $\boldsymbol{\alpha}^k$ admits fast evaluation. This is not the case for the RS code basis. The vector with a similar role in the RS code basis is $(h^i)_{h\in H}$ corresponding to the monomial $f(X) := X^i$. Moreover, the vector $\left(\frac{1}{\alpha-h}\right)_{h\in H}$ is also available, as the polynomial $\frac{v_H(X)\cdot v_H(\alpha)^{-1}-1}{X-\alpha}$ has fast evaluation method exploiting the structure of H, where $v_H(X)$ is the vanishing polynomial over H. We should replace POW with queries for submitting these types of vectors instead.
- In the Reed-Solomon code basis, the Hadamard product is identified with the polynomial multiplication. Therefore, the multivariate polynomials of the HAD query are no longer restricted to be quadratic. However, every degree contributes n to the maximal degree of the resulting PIOP.
- In the Reed-Solomon code basis, the INN queries can be batched together and checked by one invocation of the *univariate sumcheck* protocol [7, 16, 18]. Similar to the HAD queries, the polynomials in INN are no longer restricted to be quadratic.
- In the monomial basis, shifting a vector $\boldsymbol{v}$ effectively multiplies X^k to the polynomial $f_{\boldsymbol{v}}(X)$. In the Reed-Solomon code basis, the shifting is implemented by replacing $f(X)$ with $f(g^{-k}\cdot X)$ assuming H is a multiplicative subgroup generated by $g \in \mathbb{F}^*$. This shifting is cyclic rather than zero-padded, and the SHR query should be redefined accordingly.

We will not dive into the details and leave this alternative compiler to future work.

3.3 The VOProof Workflow

With the VO model and VO-to-PIOP compiler, the designers are now ready to construct zkSNARKs via the VOProof workflow illustrated in Fig. 4. In this workflow, the designer *needs only to manually accomplish the second step* (i.e., the VO protocol design), and the other steps are completed *automatically* by existing tools. VOProof thus facilitates simpler zkSNARK designs than prior works.

Next, we apply this workflow to construct zkSNARKs for popular constraint systems. The simplicity of VO protocols allows us to present three zkSNARKs in one section, whereas each of them might require an entire paper presented in traditional methods.

4 ZKSNARKS FROM VOPROOF

We construct zkSNARKs for three circuit-based constraint systems: R1CS, HPR and PLONK defined in Equations (9), (10), (11), respectively. We present the VO protocols for these relations, namely VOR1CS[6], VOHPR and VOPLONK. Afterward, these VO protocols can be compiled to zkSNARKs via the compilers.

[6]We also designed a VO protocol for R1CS-Lite, a more lightweight variant of R1CS, proposed by Lunar [14]. We leave it in Appendix A.

[scale=0.8](node0)$y = F(x, w)$; [scale=0.8,right=0.6cm of node0](node1)Constraint System; [scale=0.8,right=0.6cm of node1](node2)VO Protocol; [scale=0.8,right=0.6cm of node2](node3)PIOP; [scale=0.8,right=0.6cm of node3](node4)zkSNARK; `draw,-stealth](node0)` `tonodeabove,midway,sloped,scale=0.7,above=0.2cm]`arithmetization (node1); `draw,-stealth](node1)` `tonodeabove,midway,sloped,scale=0.7,above=0.2cm]`protocol design (node2); `draw,-stealth](node2)` `tonodeabove,midway,sloped,scale=0.7,above=0.2cm]`vo compile (node3); `draw,-stealth](node3)` `tonodeabove,midway,sloped,scale=0.7,above=0.2cm]`crypto compile (node4);

Figure 4: The VOProof zkSNARK construction workflow.

Note that both R1CS and HPR involve matrix-vector multiplications, where the matrix is usually sparse in practice. Therefore, we develop the SMVP (Sparse Matrix-Vector Product) protocol as a building block shared by VOR1CS and VOHPR. Regarding the PLONK relation in Equation (11), we remark that this version modifies the original constraint system in the PLONK paper [21] for reasons explained in Sect. 4.2. The SMVP protocol and the PLONK relation are the most complex constructions in this section, so we provide high-level overviews to highlight the ideas behind them. The VO protocols for VOR1CS, VOHPR and VOPLONK are straightforward combinations of the building blocks.

$$\mathcal{R}_{\mathsf{R1CS}} = \left\{ \left(\begin{pmatrix} H, K, \ell \\ \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \end{pmatrix}, \begin{matrix} \boldsymbol{x}, \\ \boldsymbol{w} \end{matrix} \right) \middle| \begin{array}{l} \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{H\times K} \\ \boldsymbol{x} \in \mathbb{F}^{\ell}, \boldsymbol{w} \in \mathbb{F}^{K-\ell-1} \\ (\boldsymbol{A}\boldsymbol{z}) \circ (\boldsymbol{B}\boldsymbol{z}) = \boldsymbol{C}\boldsymbol{z} \\ \quad \text{where } \boldsymbol{z} = 1\|\boldsymbol{x}\|\boldsymbol{w} \end{array} \right\} \tag{9}$$

$$\mathcal{R}_{\mathsf{HPR}} = \left\{ \left(\begin{pmatrix} H, K, \ell \\ \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{d} \end{pmatrix}, \begin{matrix} \boldsymbol{x}, \\ \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3 \end{matrix} \right) \middle| \begin{array}{l} \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{H\times K}, \boldsymbol{d} \in \mathbb{F}^{H} \\ \boldsymbol{x} \in \mathbb{F}^{\ell}, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3 \in \mathbb{F}^{K} \\ \boldsymbol{w}_1 \circ \boldsymbol{w}_2 = \boldsymbol{w}_3 \\ \boldsymbol{A}\boldsymbol{w}_1 + \boldsymbol{B}\boldsymbol{w}_2 + \boldsymbol{C}\boldsymbol{w}_3 + \boldsymbol{d} \\ \quad = \boldsymbol{x}\|\boldsymbol{0}^{H-\ell} \end{array} \right\} \tag{10}$$

$$\mathcal{R}_{\mathsf{PLK}} = \left\{ \left(\begin{pmatrix} C_c, C_a, \\ C_m, \boldsymbol{d}, \\ \sigma, \mathcal{I}_x \end{pmatrix} \begin{matrix} \boldsymbol{x}, \\ (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) \end{matrix} \right) \middle| \begin{array}{l} C := C_a + C_m \\ \sigma \in \Sigma([3C + C_c]) \\ \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{F}^{C}, \boldsymbol{x} \in \mathbb{F}^{3C}, \boldsymbol{d} \in \mathbb{F}^{C_c} \\ \boldsymbol{a}_{[1..C_m]} \circ \boldsymbol{b}_{[1..C_m]} = \boldsymbol{c}_{[1..C_m]} \\ \boldsymbol{a}_{[C_m+1..C]} + \boldsymbol{b}_{[C_m+1..C]} = \\ \quad \boldsymbol{c}_{[C_m+1..C]} \\ (\boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c})_{[i]} = \boldsymbol{x}_{[i]} \text{ for } i \in \mathcal{I}_x \\ \sigma(\boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c}\|\boldsymbol{d}) = \boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c}\|\boldsymbol{d} \end{array} \right\} \tag{11}$$

We present the VO protocols for the above relations in Fig. 5.

4.1 Overview of SMVP Protocol

Let $\boldsymbol{M} \in \mathbb{F}^{H\times K}$ be a sparse matrix with at most S nonzero elements. Given the handles to vectors $\boldsymbol{a}, \boldsymbol{b}$, the protocol SMVP allows the verifier to check that $\boldsymbol{b}_{[1..H]} = \boldsymbol{M}\boldsymbol{a}_{[1..K]}$. We highlight the key ideas as follows:

Random linear combination. We multiply a random vector $\boldsymbol{r}_\alpha$ with special structure (defined in step 3 of Fig. 5) to both sides of the equation. By Lemma 4.1, the equality $\boldsymbol{r}_\alpha{}^{\mathsf{T}}\boldsymbol{b}_{[1..H]} = \boldsymbol{r}_\alpha{}^{\mathsf{T}}\boldsymbol{M}\boldsymbol{a}_{[1..K]}$

procedure $\mathsf{VOR1CS}_{\mathsf{offline}}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$
  Run $\mathsf{SMVP}_{\mathsf{offline}}(\boldsymbol{M} = (\boldsymbol{A}^\top \| \boldsymbol{C}^\top \| \boldsymbol{B}^\top)^\top \in \mathbb{F}^{3H \times K})$.

procedure $\mathsf{VOR1CS}_{\mathsf{online}}(\boldsymbol{x}; \boldsymbol{w})$
  P submits $\boldsymbol{w}$ and $\boldsymbol{y} := \boldsymbol{M}(1 \| \boldsymbol{x} \| \boldsymbol{w})$ and sends $h_{\boldsymbol{w}}, h_{\boldsymbol{y}}$ to V;
  Run $\mathsf{SMVP}_{\mathsf{online}}(h_{1\|\boldsymbol{x}\|\boldsymbol{w}}, h_{\boldsymbol{y}}; 1 \| \boldsymbol{x} \| \boldsymbol{w}, \boldsymbol{y})$;
  V checks $\boldsymbol{y} \circ \boldsymbol{y}^{\to 2H} \stackrel{?}{=}_n \boldsymbol{1}_{[2H+1..3H]} \circ \boldsymbol{y}^{\to H}$.

procedure $\mathsf{VOHPR}_{\mathsf{offline}}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{d})$
  Run $\mathsf{SMVP}_{\mathsf{offline}}(\boldsymbol{M} = (\boldsymbol{A} \| \boldsymbol{C} \| \boldsymbol{B}) \in \mathbb{F}^{H \times (3K+1)})$.

procedure $\mathsf{VOHPR}_{\mathsf{online}}(\boldsymbol{x}; \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$
  P submits $\boldsymbol{w} := \boldsymbol{w}_1 \| \boldsymbol{w}_3 \| \boldsymbol{w}_2$ and sends $h_{\boldsymbol{w}}$ to V;
  Run $\mathsf{SMVP}(h_{1\|\boldsymbol{w}}, h_{\boldsymbol{x}}; 1 \| \boldsymbol{w}, \boldsymbol{x})$;
  V checks $\boldsymbol{w} \circ \boldsymbol{w}^{\to 2K} \stackrel{?}{=}_n \boldsymbol{1}_{[2K+1..3K]} \circ \boldsymbol{w}^{\to K}$.

procedure $\mathsf{VOPLONK}_{\mathsf{offline}}(C_c, C_a, C_m, \boldsymbol{d}, \sigma, \mathcal{I}_x)$
  I submits $\boldsymbol{d}, \boldsymbol{\sigma} := \left(\gamma^{\sigma(i)-1}\right)_{i=1}^{3C+C_c}$ where $C = C_a + C_m$;
  I sends $\boldsymbol{d}, \boldsymbol{\sigma}$ to P, and $h_{\boldsymbol{d}}, h_{\boldsymbol{\sigma}}$ to V.

procedure $\mathsf{VOPLONK}_{\mathsf{online}}(\boldsymbol{x}; \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$
  P submits $\boldsymbol{u} := \boldsymbol{a} \| \boldsymbol{b}_{[1..C_m]}, \boldsymbol{v} := \boldsymbol{b}_{[C_m+1..C]} \| \boldsymbol{c}$ and sends $h_{\boldsymbol{u}}, h_{\boldsymbol{v}}$ to V;
  V checks $\boldsymbol{u}^{\to C} \circ \boldsymbol{u} \stackrel{?}{=}_n \boldsymbol{v}^{\to C_m} \circ \boldsymbol{1}_{[C+1..C+C_m]}$;
  V checks $(\boldsymbol{u}^{\to C_a} + \boldsymbol{v}^{\to C} - \boldsymbol{v}) \circ \boldsymbol{1}_{[C+1..C+C_a]} \stackrel{?}{=}_n \boldsymbol{0}$;
  Run $\mathsf{VBatch}(h_{\boldsymbol{u}\|\boldsymbol{v}}, \{i, \boldsymbol{x}_{[i]}\}_{i \in \mathcal{I}_x}; \boldsymbol{u} \| \boldsymbol{v})$;
  V samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends β to P;
  V submits $\boldsymbol{w} := \boldsymbol{u} \| \boldsymbol{v} \| \boldsymbol{d}$, $\boldsymbol{s} := \boldsymbol{\sigma} + \beta \cdot \boldsymbol{w}$, $\boldsymbol{t} := \boldsymbol{\gamma}^{3C+C_c} + \beta \cdot \boldsymbol{w}$;
  Run $\mathsf{PermCheck}(h_{\boldsymbol{t}}, h_{\boldsymbol{s}}, 3C + C_c; \boldsymbol{t}, \boldsymbol{s})$.

procedure $\mathsf{SMVP}_{\mathsf{offline}}(\boldsymbol{M})$
  Assume that $\boldsymbol{M} \in \mathbb{F}^{H \times K}$ has S nonzero entries with row indices $\{\mathsf{row}_i \in [H]\}_{i=1}^S$, column indices $\{\mathsf{col}_i \in [K]\}_{i=1}^S$ and values $\{\mathsf{val}_i \in \mathbb{F}\}_{i=1}^S$;
  Let γ be the generator of the multiplicative group $\mathbb{F}^*$;
  I submits $\boldsymbol{u} = (\gamma^{\mathsf{row}_i})_{i=1}^S$, $\boldsymbol{w} = \left(\gamma^{\mathsf{col}_i}\right)_{i=1}^S$, $\boldsymbol{v} = (\mathsf{val}_i)_{i=1}^S$ and $\boldsymbol{y} = \boldsymbol{u} \circ \boldsymbol{w}$;
  I sends $\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v}, \boldsymbol{y}$ to P, and $h_{\boldsymbol{u}}, h_{\boldsymbol{w}}, h_{\boldsymbol{v}}, h_{\boldsymbol{y}}$ to V.

1: **procedure** $\mathsf{SMVP}_{\mathsf{online}}(h_{\boldsymbol{a}}, h_{\boldsymbol{b}}; \boldsymbol{a}, \boldsymbol{b})$
2: V samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends α to P;
3: P submits $\boldsymbol{r}_\alpha := \left(\frac{1}{\alpha - \gamma^i}\right)_{i=1}^H$ and $\boldsymbol{c} := \boldsymbol{r}_\alpha^\top \boldsymbol{M}$ and sends $h_{\boldsymbol{r}_\alpha}, h_{\boldsymbol{c}}$ to V;
4: V checks $\boldsymbol{r}_\alpha \circ (\alpha \cdot \boldsymbol{1}^n - \boldsymbol{\gamma}^H) \stackrel{?}{=}_n \boldsymbol{1}^H$;
5: V checks $\langle \boldsymbol{r}_\alpha, \boldsymbol{b} \rangle \stackrel{?}{=} \langle \boldsymbol{c}, \boldsymbol{a} \rangle$ and $\boldsymbol{c} \circ \left(\boldsymbol{1}^{n-K}\right)^{\to K} \stackrel{?}{=}_n \boldsymbol{0}$;
6: V samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends β to P;
7: P submits $\boldsymbol{r}_\beta := \left(\frac{1}{\beta - \gamma^i}\right)_{i=1}^K$ and $\boldsymbol{t} := \left(\frac{1}{(\alpha - \gamma^{\mathsf{row}_i})(\beta - \gamma^{\mathsf{col}_i})}\right)_{i=1}^S$ and sends $h_{\boldsymbol{r}_\beta}, h_{\boldsymbol{t}}$ to V;
8: V checks $\boldsymbol{r}_\beta \circ (\beta \cdot \boldsymbol{1}^n - \boldsymbol{\gamma}^K) \stackrel{?}{=}_n \boldsymbol{1}^K$;
9: V checks $\boldsymbol{t} \circ (\alpha\beta \cdot \boldsymbol{1}^n - \alpha \cdot \boldsymbol{w} - \beta \cdot \boldsymbol{u} + \boldsymbol{y}) \stackrel{?}{=}_n \boldsymbol{1}^S$;
10: V checks $\langle \boldsymbol{r}_\beta, \boldsymbol{c} \rangle \stackrel{?}{=} \langle \boldsymbol{t}, \boldsymbol{v} \rangle$.

Figure 5: VO Protocols for General-Purpose zkSNARKs.

suffices to guarantee that the original equation holds. To check this new equation, the prover submits $\boldsymbol{c} := \boldsymbol{r}_\alpha{}^\top \boldsymbol{M} \in \mathbb{F}^K$ and sends $h_{\boldsymbol{c}}$ to the verifier. By substituting $\boldsymbol{c}$, the new equation becomes $\langle \boldsymbol{r}_\alpha{}^\top, \boldsymbol{b} \rangle = \langle \boldsymbol{c}, \boldsymbol{a} \rangle$. After validating $\boldsymbol{r}_\alpha$ (step 5) and this new equation (step 6), the problem is reduced to validating $\boldsymbol{c}$, i.e., the equation $\boldsymbol{c} = \boldsymbol{r}_\alpha{}^\top \boldsymbol{M} \in \mathbb{F}^K$.

We repeat the above procedure by similarly multiplying vector $\boldsymbol{r}_\beta$ to both sides of the equation. After validating $\boldsymbol{r}_\beta$ (step 10), the problem is finally reduced to verifying $\langle \boldsymbol{c}, \boldsymbol{r}_\beta \rangle = \boldsymbol{r}_\alpha{}^\top \boldsymbol{M} \boldsymbol{r}_\beta$, which we deal with next.

Sparse Representation of Matrix. Assume the matrix $\boldsymbol{M}$ has S nonzero entries, with row indices $(\mathsf{row}_1, \cdots, \mathsf{row}_S) \in [H]^S$, column indices $(\mathsf{col}_1, \cdots, \mathsf{col}_S) \in [K]^S$ and values $\boldsymbol{v} \in \mathbb{F}^S$. By the definition of $\boldsymbol{r}_\alpha, \boldsymbol{r}_\beta$ and matrix-vector product,

$$\boldsymbol{r}_\alpha{}^\top \boldsymbol{M} \boldsymbol{r}_\beta = \sum_{i,j=1}^{H,K} \boldsymbol{M}_{[i,j]} \cdot \boldsymbol{r}_{\alpha[i]} \cdot \boldsymbol{r}_{\beta[j]} = \sum_{i \in [S]} \frac{\boldsymbol{v}_{[i]}}{(\alpha - \gamma^{\mathsf{row}_i})(\beta - \gamma^{\mathsf{col}_i})}.$$

This is the inner product between $\boldsymbol{v}$ and $\boldsymbol{t} := \left(\frac{1}{(\alpha - \gamma^{\mathsf{row}_i})(\beta - \gamma^{\mathsf{col}_i})}\right)_{i=1}^S$. The prover submits the vector $\boldsymbol{t}$, which is validated by step 11, and the equation becomes $\langle \boldsymbol{v}, \boldsymbol{t} \rangle \stackrel{?}{=} \langle \boldsymbol{c}, \boldsymbol{r}_\beta \rangle$ which is checked in step 12.

4.2 Overview of the Modified PLONK Relation

The constraint system of PLONK, defined by the index relation $\mathcal{R}_{\mathsf{PLK}}$, is a characterization of the computations by fan-in-two circuits. We explain the transformation from fan-in-two circuits to the relation $\mathcal{R}_{\mathsf{PLK}}$ as follows.

Assume the circuit contains C gates, including C_a addition gates, C_m multiplication gates and C_c constant gates, such that $C = C_a + C_m + C_c$. For the i-th gate, denote its left input by a_i, its right input by b_i and its output by c_i. These variables satisfy the following constraints:

- **addition constraints**: let $\mathcal{I}_a \subset [C]$ be the index set for the addition gates, then $a_i + b_i = c_i$ for $i \in \mathcal{I}_a$;
- **multiplication constraints**: let $\mathcal{I}_m \subset [C]$ be the index set for the multiplication gates, then $a_i \cdot b_i = c_i$ for $i \in \mathcal{I}_m$;
- **constant constraints**: let $\mathcal{I}_c \subset [C]$ be the index set for the constant gates, and for $i \in \mathcal{I}_c$, denote the constant value of the i-th gate by d_i, which is publicly known as part of the circuit description, then $c_i = d_i$ for $i \in \mathcal{I}_c$;
- **public input/output**: let $\boldsymbol{w} := (a_i)_{i=1}^C \| (b_i)_{i=1}^C \| (c_i)_{i=1}^C \in \mathbb{F}^{3C}$ be the vector of all variables, $\mathcal{I}_x \subset [3C]$ be the indices where the variable values are publicly known, then $\boldsymbol{w}_{[i]} = x_i$ for $i \in \mathcal{I}_x$ where x_i are the publicly known values;
- **copy constraints**: let $S_1 \cup \cdots \cup S_L$ be a partition over $[3C]$ such that variables connected by wires fall into the same

partition, then for any pair of $(i, j) \in [3C]^2$ in the same partition, $\boldsymbol{w}_{[i]} = \boldsymbol{w}_{[j]}$.

PLONK collects these variables into three vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{F}^C$. To characterize the above constraints, PLONK introduces five vectors $\boldsymbol{q_L}$, $\boldsymbol{q_R}$, $\boldsymbol{q_O}$, $\boldsymbol{q_M}$ and $\boldsymbol{q_C}$. By properly setting the values of these five vectors, the first three types of constraints are equivalent to

$$\boldsymbol{a} \circ \boldsymbol{q_L} + \boldsymbol{b} \circ \boldsymbol{q_R} + \boldsymbol{c} \circ \boldsymbol{q_O} + \boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{q_M} + \boldsymbol{q_C} = \boldsymbol{0}. \tag{12}$$

PLONK handles the public input/output constraints by dummy gates, which we omit here for simplicity. For the copy constraints, PLONK relies on the permutation check, which we briefly describe as follows.

Let $\boldsymbol{v} \in \mathbb{F}^{3C}$ be a vector. For any permutation σ over $[3C]$, and any $i \in [3C]$, by starting from i and repeatedly applying σ, we can get a sequence of integers $i, \sigma(i), \cdots, \sigma^{-1}(i), i$ that cycles back to i. All such cycles form a partition over $[3C]$. For any partition $S_1 \cup \cdots \cup S_L$ over $[3C]$, we can find a σ whose cycles exactly form this partition. It is straightforward to check that if $\boldsymbol{v}$ is invariant under the permutation σ, i.e., $\boldsymbol{v} = \sigma(\boldsymbol{v}) := (\boldsymbol{v}_{[\sigma(i)]})_{i=1}^n$, then $\boldsymbol{v}$ satisfies the copy constraints for the cycle partition of σ.

We optimize the PLONK relation to better suit the VO model. First, we sort the gates in the circuit by their types. Specifically, we index the multiplication gates by $[1..C_m]$, the addition gates by $[C_m+1..C_m+C_a]$, and the constant gates by $[C_m+C_a+1..C]$. As a result of sorting, the vectors $\boldsymbol{q_L}, \boldsymbol{q_R}, \boldsymbol{q_O}$ and $\boldsymbol{q_M}$ become consecutive ones and can be eliminated from the index, and $\boldsymbol{q_C}$ becomes a shorter vector, which we denote by $\boldsymbol{d} \in \mathbb{F}^{C_c}$. Moreover, the last C_c elements of both $\boldsymbol{a}$ and $\boldsymbol{b}$ (corresponding to the inputs to constant gates) are irrelevant to the circuit, and the last C_c elements of $\boldsymbol{c}$ are exactly $\boldsymbol{d}$. Therefore, we can redefine the witness vectors $\boldsymbol{a}, \boldsymbol{b}$, and $\boldsymbol{c}$ to contain only the outputs of the addition and multiplication gates. For simplicity of notation, we also redefine C as $C_a + C_m$ instead of $C_a + C_m + C_c$ in the relation and the protocol. The modified PLONK relation is summarized in Equation (11).

4.3 Security

We show that the protocols in Fig. 5 have negligible soundness error. We start from introducing some necessary lemmas and definitions.

Lemma 4.1. *Let γ be a generator of the multiplicative group $\mathbb{F}^*$. For any nonzero vector $\boldsymbol{v} \in \mathbb{F}^H$, for uniformly random $\alpha \in \mathbb{F} \backslash \{\gamma^i\}_{i=1}$, the probability* $\Pr[\langle \boldsymbol{r}_\alpha, \boldsymbol{v} \rangle = 0] \le \frac{H}{|\mathbb{F}|-|H|}$, *where* $\boldsymbol{r}_\alpha := \left(\frac{1}{\alpha-\gamma^i}\right)_{i=1}^H$.

Proof. Note that

$$\sum_{i=1}^{H} \frac{\boldsymbol{v}_{[i]}}{\alpha - \gamma^i} = 0 \Leftrightarrow \quad \sum_{i=1}^{H} \boldsymbol{v}_{[i]} \prod_{j \ne i} (\alpha - \gamma^j) = 0 \Leftrightarrow \quad \boldsymbol{v}^\top \boldsymbol{\Gamma} \boldsymbol{\alpha}^H = 0,$$

where $\boldsymbol{\Gamma}$ is a matrix of size $H \times H$ whose i-th row is the coefficient vector of polynomial $\prod_{j \ne i}(X - \gamma^j)$, which (after normalized) is the Lagrange basis polynomial over $\{\gamma^i\}_{i=1}^H$. Since the Lagrange basis polynomials are linearly independent, $\boldsymbol{\Gamma}$ is an invertible matrix, therefore $\boldsymbol{v}^\top \boldsymbol{\Gamma} \ne \boldsymbol{0}$. Since $\boldsymbol{v}^\top \boldsymbol{\Gamma} \boldsymbol{\alpha}^H = f_{\boldsymbol{v}^\top \boldsymbol{\Gamma}}(\alpha)$, the conclusion follows from Schwartz-Zippel Lemma.

□

Lemma 4.2 (restate of Lemma A.3 of [21]). *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^\ell$. If $\boldsymbol{u}$ and $\boldsymbol{v}$ are not permutations of each other, then for uniformly random $\alpha \in \mathbb{F}$, the probability that $\prod_{i \in [\ell]}(\boldsymbol{u}_{[i]} + \alpha) = \prod_{i \in [\ell]}(\boldsymbol{v}_{[i]} + \alpha)$ is bounded by $\frac{\ell}{|\mathbb{F}|}$.*

Definition 4.3 (Simultaneous Permutation). Let $\{\boldsymbol{u}^{(j)}\}_{j=1}^m, \{\boldsymbol{v}^{(j)}\}_{j=1}^m$ be two groups of vectors in $\mathbb{F}^\ell$. We say they are simultaneous permutations of each other, denoted by $\{\boldsymbol{u}^{(j)}\}_{j=1}^m \sim \{\boldsymbol{v}^{(j)}\}_{j=1}^m$, if there exists a permutation σ over $[\ell]$ such that $\boldsymbol{u}^{(j)}_{[\sigma(i)]} = \boldsymbol{v}^{(j)}_{[i]}$ for any $i \in [\ell]$ and $j \in [m]$.

Lemma 4.4. *Let $\{\boldsymbol{u}^{(j)}\}_{j=1}^m, \{\boldsymbol{v}^{(j)}\}_{j=1}^m$ be two groups of vectors in $\mathbb{F}^\ell$. If $\{\boldsymbol{u}^{(j)}\}_{j=1}^m \nsim \{\boldsymbol{v}^{(j)}\}_{j=1}^m$, then for uniformly random $\alpha \in \mathbb{F}$, the probability that $\sum_{j=1}^m \alpha^{j-1}\boldsymbol{u}^{(j)} \sim \sum_{j=1}^m \alpha^{j-1}\boldsymbol{v}^{(j)}$ is bounded by $\frac{2(m-1)\ell}{|\mathbb{F}|}$.*

Proof. Consider two sequences $U := \{(\boldsymbol{u}^{(1)}_{[i]}, \cdots, \boldsymbol{u}^{(m)}_{[i]})\}_{i \in [\ell]}$ and $V := \{(\boldsymbol{v}^{(1)}_{[i]}, \cdots, \boldsymbol{v}^{(m)}_{[i]})\}_{i \in [\ell]}$. Since $\{\boldsymbol{u}^{(j)}\}_{j=1}^m \nsim \{\boldsymbol{v}^{(j)}\}_{j=1}^m$, obviously U and V are not permutations of each other. Therefore, there exists some tuple $(a_1, a_2, \cdots, a_m)$ that appears different number of times in U and V. Let $\mathcal{I}_u$ be the index set where $(\boldsymbol{u}^{(1)}_{[i]}, \cdots, \boldsymbol{u}^{(m)}_{[i]}) = (a_1, a_2, \cdots, a_m)$ and $\mathcal{I}_v$ be the similarly defined for V. Then $|\mathcal{I}_u| \ne |\mathcal{I}_v|$. For any tuple $(b_1, b_2, \cdots, b_m) \ne (a_1, a_2, \cdots, a_m)$, for uniformly random α over $\mathbb{F}$, the probability that $\sum_{j=1}^m \alpha^{j-1} a_j = \sum_{j=1}^m \alpha^{j-1} b_j$ is bounded by $\frac{m-1}{|\mathbb{F}|}$, due to Schwartz-Zippel Lemma. By the union bound, the probability that $\sum_{j=1}^m \alpha^{j-1} a_j = \sum_{j=1}^m \alpha^{j-1}\boldsymbol{u}^{(j)}_{[i]}$ for any $i \notin \mathcal{I}_u$ or $\sum_{j=1}^m \alpha^{j-1} a_j = \sum_{j=1}^m \alpha^{j-1}\boldsymbol{v}^{(j)}_{[i]}$ for any $i \notin \mathcal{I}_v$ is less than $\frac{2(m-1)\ell}{|\mathbb{F}|}$.

Therefore, except with probability $\frac{2(m-1)\ell}{|\mathbb{F}|}$, the value $\sum_{j=1}^m \alpha^{j-1} a_j$ appears exactly $|\mathcal{I}_u|$ times in vector $\sum_{j=1}^m \alpha^{j-1}\boldsymbol{u}^{(j)}$ and $|\mathcal{I}_v|$ (i.e. $\ne |\mathcal{I}_u|$) times in vector $\sum_{j=1}^m \alpha^{j-1}\boldsymbol{v}^{(j)}$, which ensures that the two vectors are not permutations of each other.

□

Lemma 4.5. *The* SMVP *protocol in Fig. 5 has perfect completeness and soundness error $\frac{H+K}{|\mathbb{F}|-H-K}$.*

Proof. *Completeness.* If the prover is honest and $\boldsymbol{b}_{[1..H]} = \boldsymbol{M}\boldsymbol{a}_{[1..K]}$, then $\boldsymbol{r}_\alpha, \boldsymbol{r}_\beta, \boldsymbol{t}$ are computed as defined and the checks in step 5, 10 and 11 will pass. Moreover, $\boldsymbol{c} = \boldsymbol{r}_\alpha^\top \boldsymbol{M}$ and the correctness of $\boldsymbol{t}$ imply that step 6 and 12 will pass.

Soundness. If $\boldsymbol{b}_{[1..H]} = \boldsymbol{M}\boldsymbol{a}_{[1..K]}$, consider the strategy of a malicious prover. Note that the prover must submit $\boldsymbol{r}_\alpha, \boldsymbol{r}_\beta$ and $\boldsymbol{t}$ correctly, otherwise step 5, 10 and 11 directly fail. If the prover submits the correct $\boldsymbol{c} = \boldsymbol{r}_\alpha^\top \boldsymbol{M}$, then except with probability $H/(|\mathbb{F}| - H)$, $\boldsymbol{r}_\alpha^\top \boldsymbol{M}\boldsymbol{a} \ne \boldsymbol{r}_\alpha^\top \boldsymbol{a}$ and step 6 fails, according to Lemma 4.1. Otherwise, the prover submits an incorrect $\boldsymbol{c}$. If $\boldsymbol{c}$ contains nonzero elements in positions after K, then step 6 will also fail (at the second check). So $\boldsymbol{c}$ must differ from the correct $\boldsymbol{c}$ at one of the first K positions. Then by Lemma 4.1 again, the inner product of $\boldsymbol{c}$ and $\boldsymbol{r}_\beta$ differs from $\boldsymbol{r}_\alpha^\top \boldsymbol{M}\boldsymbol{r}_\beta$, which equals the inner product of $\boldsymbol{t}$ and $\boldsymbol{v}$, except with probability $K/(|\mathbb{F}| - K)$, which is also the probability that step 12 passes. In conclusion, the soundness error is bounded by $(H+K)/(|\mathbb{F}| - \max\{H, K\}) \le (H+K)/(|\mathbb{F}| - H - K)$.

□

Theorem 4.6. *The* VOR1CS *protocol in Fig. 5 is a VO protocol for the relation* $\mathcal{R}_{\mathsf{R1CS}}$ *with perfect completeness and soundness error* $\frac{3H+K}{|\mathbb{F}|-3H-K}$.

Proof. *Completeness.* If the input is a correct instance of R1CS, and $\boldsymbol{y}$ is computed honestly, then $\boldsymbol{y}_{[1..H]} \circ \boldsymbol{y}_{[2H+1..3H]} = \boldsymbol{y}_{[H+1..2H]}$, and the SMVP protocol will pass due to completeness of SMVP. The last check also passes because boths sides have $\boldsymbol{y}_{[H+1..2H]}$ at the positions $[2H+1..3H]$ and zero elsewhere.

Soundness. If the input is incorrect, then either $\boldsymbol{y}_{[1..H]} \circ \boldsymbol{y}_{[2H+1..3H]} \neq \boldsymbol{y}_{[H+1..2H]}$ or $\boldsymbol{y} \neq \boldsymbol{M}(1\|\boldsymbol{x}\|\boldsymbol{w})$. In the first case, the last check fails. In the second case, the SMVP protocol rejects except with probability $(3H+K)/(|\mathbb{F}|-3H-K)$.

□

Theorem 4.7. *The* VOHPR *protocol in Fig. 5 is a VO protocol for the relation* $\mathcal{R}_{\mathsf{HPR}}$ *with perfect completeness and soundness error* $\frac{3K+H+1}{|\mathbb{F}|-H-3K-1}$.

Proof. *Completeness.* If the input is a correct instance of R1CS, then $\boldsymbol{w}_{[1..K]} \circ \boldsymbol{w}_{[2K+1..3K]} = \boldsymbol{w}_{[K+1..2K]}$, and the SMVP protocol will pass due to completeness of SMVP. The last check also passes because boths sides have $\boldsymbol{w}_{[K+1..2K]}$ at the positions $[2K+1..3K]$ and zero elsewhere.

Soundness. If the input is incorrect, then either $\boldsymbol{w}_{[1..K]} \circ \boldsymbol{w}_{[2K+1..3K]} \neq \boldsymbol{w}_{[K+1..2K]}$ or $\boldsymbol{x}\|\boldsymbol{0} \neq \boldsymbol{M}(1\|\boldsymbol{w})$. In the first case, the last check fails. In the second case, the SMVP protocol rejects except with probability $(3K+H+1)/(|\mathbb{F}|-3K-H-1)$.

□

Lemma 4.8. *The* ProdCheck *protocol in Fig. 2 is perfectly complete when* $\boldsymbol{u}_{[1..\ell]}$ *and* $\boldsymbol{v}_{[1..\ell]}$ *do not contain any zero elements, and perfectly sound.*

Proof. *Completeness.* When $0 \neq \prod_{i\in[\ell]} \boldsymbol{u}_{[i]} = \prod_{i\in[\ell]} \boldsymbol{v}_{[i]}$, the vector $\boldsymbol{r}$ is well-defined and $\boldsymbol{r}_{[\ell]} = 1$, so the last check passes. By the definition of $\boldsymbol{r}$, $\boldsymbol{r}_{[i]} \cdot \boldsymbol{v}_{[i]} = \boldsymbol{r}_{[i-1]} \cdot \boldsymbol{u}_{[i]}$ for every $i \in [2..\ell]$, and $\boldsymbol{r}_{[1]} \cdot \boldsymbol{v}_{[1]} = 1 \cdot \boldsymbol{u}_{[1]}$. Therefore, the verifier accepts with probability 1. Note that the right shift by $n-\ell$ excludes the elements after position ℓ from the check.

Soundness. If $\prod_{i\in[\ell]} \boldsymbol{u}_{[i]} \neq \prod_{i\in[\ell]} \boldsymbol{v}_{[i]}$, then $\boldsymbol{r}_{[\ell]} \neq 1$. Therefore, the prover either submits an incorrect $\boldsymbol{r}$ and fails at the first HAD query or submits the correct $\boldsymbol{r}$ and fails at the second check.

□

Lemma 4.9. *The* PermCheck *protocol in Fig. 2 has completeness error* $\frac{\ell}{|\mathbb{F}|}$, *soundness error* $\frac{3\ell}{|\mathbb{F}|}$.

Proof. *Completeness.* If $\boldsymbol{u}_{[1..\ell]} \sim \boldsymbol{v}_{[1..\ell]}$ then $\prod_{i=1}^{\ell}(\boldsymbol{u}_{[i]}+\alpha) = \prod_{i=1}^{\ell}(\boldsymbol{v}_{[i]}+\beta)$. Except with probability $\frac{\ell}{|\mathbb{F}|-1}$, $-\beta \notin \boldsymbol{u}_{[1..\ell]}$, which is equivalent to $-\beta \notin \boldsymbol{v}_{[1..\ell]}$. By the perfect completeness of ProdCheck protocol, the completeness error of PermCheck protocol is $\frac{\ell}{|\mathbb{F}|-1}$.

Soundness. If $\boldsymbol{u}_{[1..\ell]} \nsim \boldsymbol{v}_{[1..\ell]}$, then by Lemma 4.2, $\prod_{i=1}^{\ell}(\boldsymbol{u}_{[i]}+\beta) \neq \prod_{i=1}^{\ell}(\boldsymbol{v}_{[i]}+\beta)$ except with probability $\frac{\ell}{|\mathbb{F}|}$. Moreover, except with probability $\frac{2\ell}{|\mathbb{F}|-1}$, $-\beta \notin \boldsymbol{u}_{[1..\ell]} \cup \boldsymbol{v}_{[1..\ell]}$. By the perfect soundness of ProdCheck protocol, the soundness error of PermCheck is bounded by $\frac{3\ell}{|\mathbb{F}|}$.

□

Theorem 4.10. *The* VOPLONK *protocol in Fig. 5 is a VO protocol that validates the relation* $\mathcal{R}_{\mathsf{PLK}}$ *with completeness error* $\frac{3C+C_c}{|\mathbb{F}|}$, *soundness error* $\frac{15C+5C_c}{|\mathbb{F}|}$.

Proof. *Completeness.* Note that $\boldsymbol{u}\|\boldsymbol{v}$ is simply $\boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c}$. If $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ are valid witnesses, the first C_m elements of $\boldsymbol{u}$ multiply the last C_m elements of $\boldsymbol{u}$ should equal the middle C_m elements of $\boldsymbol{v}$, and the middle C_a elements of $\boldsymbol{u}$ plus the first C_a elements of $\boldsymbol{v}$ should equal the last C_a elements of $\boldsymbol{v}$, hence the first two HAD queries. We have $(\boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c})_{[i]} = \boldsymbol{x}_{[i]}$ for every $i \in \mathcal{I}_x$, so the VBatch protocol will succeed. Finally, since $\boldsymbol{w} := \boldsymbol{a}\|\boldsymbol{b}\|\boldsymbol{c}\|\boldsymbol{d}$ is invariant under the permutation σ, the vector pairs $(\boldsymbol{\gamma}^{3C+C_c}, \boldsymbol{w})$ and $(\boldsymbol{\sigma}, \boldsymbol{w})$ are simultaneous permutations of each other by σ. Then $\boldsymbol{t} := \boldsymbol{\gamma}^{3C+C_c} + \beta \cdot \boldsymbol{w}$ and $\boldsymbol{s} := \boldsymbol{\sigma} + \beta \cdot \boldsymbol{w}$ are permutations of each other by σ. By completeness of the PermCheck protocol, the completeness error of VOPLONK is $3C + C_c$.

Soundness. If $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ are not valid witnesses, they fail at least one of the addition condition, the multiplication condition, the consistence with public inputs, or the permutation check. Failing any of the first three conditions directly leads to rejection. If the last condition does not hold, then the vector pairs $(\boldsymbol{\gamma}^{3C+C_c}, \boldsymbol{w})$ and $(\boldsymbol{\sigma}, \boldsymbol{w})$ are not simultaneous permutations of each other. By Lemma 4.4, the probability that $\boldsymbol{t}$ and $\boldsymbol{s}$ are permutations of each other is bounded by $2(3C+C_c)/|\mathbb{F}|$. In case that $\boldsymbol{t}$ and $\boldsymbol{s}$ are not permutations of each other, soundness of PermCheck implies that the probability of verifier acceptance is bounded by $3(3C+C_c)/|\mathbb{F}|$. By union bound, the soundness error of VOPLONK protocol is bounded by $(15C+5C_c)/|\mathbb{F}|$.

□

5 IMPLEMENTATION AND COMPARISON

To evaluate the feasibility of the VOProof method and the performance of VOProof-based zkSNARKs, we implement, in Python, a framework for describing VO protocols, and a *VO compiler*[7] that transforms the VO protocols into the fully functional code of zkSNARKs in Rust[8]. This VO compiler is the combination of our VO-to-PIOP compiler with the batched version of the KZG polynomial commitment scheme [24] proposed in PLONK [21]. Our compiler relies on the `Sympy` library for symbolic evaluations. The Rust code of the compiled zkSNARKs uses the finite fields and elliptic curves implemented by the `arkworks`[9] team.

Our implementation applies several optimizations to the compiler given in Sect. 3.2. We highlight the most important optimizations as follows.

5.1 Optimizations

Homomorphic combination of polynomial commitments. Inspired by a technique in PLONK (attributed to Mary Maller in Sect. 4 of [21]), we reduce the number of evaluation queries exploiting the additive homomorphism of KZG. As a showcase, consider checking the polynomial identity $f_1(\omega X^{-1}) \cdot g_1(X) - f_2(\omega X^{-1}) \cdot g_2(X) = h(X)$ at a random point z. The verifier first queries $f_1(X)$ and $f_2(X)$ with $\omega \cdot z^{-1}$ and receives y_1 and y_2, then computes the polynomial

[7] https://github.com/yczhangsjtu/voproof-scripts
[8] https://github.com/yczhangsjtu/voproof
[9] https://github.com/arkworks-rs

commitment for $g(X) = y_1 \cdot g_1(X) - y_2 \cdot g_2(X) - h(X)$ by linearly combining the commitments of $g_1(X), g_2(X)$ and $h(X)$. The verifier then queries $g(X)$ at z and check if the result is 0. In this example, the optimization saves 2 queries compared to naïvely querying each of $g_1(X), g_2(X)$ and $h(X)$.

Reducing the number of FFTs. In the last step of the Hadamard protocol in Fig. 3, the verifier needs to check the identity $h(z) = \bar{h}(\gamma z) - \bar{h}(z)$, where $h(X)$ includes the sum of terms of the form $f_{\boldsymbol{v}_i}(\omega X^{-1}) f_{\boldsymbol{v}_j}(X)$. Assume that there are m such terms. Naïvely computing $h(X)$ requires m dense polynomial multiplications that has $O(n \log n)$ complexity using FFT. However, note that each polynomial $f_{\boldsymbol{v}_i}(X)$ can be written in the form of linear combination $\sum s_i(X) f_i(X)$ where $f_i(X)$ is a polynomial sent from the prover or the indexer, and the coefficients $s_i(X)$ are *locally evaluable polynomials*—polynomials that admit fast evaluation, e.g., sparse or power polynomials. By expanding these linear combinations and collecting like terms, the prover only needs FFT in multiplications between prover-polynomials. The number of FFTs thus depends only on the number of prover-polynomials.

Splitting the polynomial $h(X)$. In the Hadamard protocol in Fig. 3, the prover can show that $h(X)$ has a zero constant term in an alternative method. This method is only efficient when combined with the homomorphic combination optimization which is not available for general PIOP. This method works by splitting the polynomial $h(X)$ into $h_1(X) \cdot X^{-D-1} + h_2(X) \cdot X$ and sending $[h_1(X)]$[10] and $[h_2(X)]$ to the verifier. The verifier checks the identity $h_1(z) \cdot z^{-D-1} + h_2(z) = h(z) \cdot z$. This optimization saves one distinct query point γz at the cost of one more online polynomial oracle, thus the proof size of the KZG polynomial commitment scheme is not affected. Meanwhile, the maximal polynomial degree is reduced from $4n$ to $2n$.

Remove unnecessary randomizations. Several vectors sent from the prover are public and do not need to be randomized for zero-knowledge. Our compiler allows the protocol designer to manually mark vectors as public and save the overhead caused by randomization.

Besides the aforementioned optimization techniques in the compiler, we also considered another technique in the VO protocol level, which we call the *vector combination*. The idea is to let the prover concatenate all the vectors that are sent in one round of interaction into a large vector. This technique reduces the proof size and the verification costs, though it sometimes causes higher setup, indexing, and proving costs, and a larger SRS size. This tradeoff is worthwhile when the verification and storage costs are more critical, e.g., in a blockchain system like Ethereum. We apply this technique to the SMVP and VOPLONK protocols manually, and leave automatic vector combination to future work. The resulting VO protocols are presented in Fig. 7 in Appendix A.

Table 1 shows the scale of the resulting PIOPs after compiling the VO protocols in the last section. We also include the numbers of Marlin, PLONK for comparison.

Applying this framework, we generate the Rust code of the zkSNARKs constructed in Sect. 4, and their verifier-efficient versions that adopt the vector combination technique. Next, we evaluate and compare these zkSNARKs with prior works in the proof size and the running time, including the setup time, the indexing (or preprocessing) time, the proving time, and the verification time.

[10]We assume here that the degree bound of PIOP is exactly D. If the degree bound is $D' > D$, then the prover should send $h_1(X) \cdot X^{D'-D}$ instead, and the verifier multiplies the query result with $z^{-D'-1}$.

Table 1: PIOP Scales. The column "#Polys" includes numbers of online and offline polynomials respectively. The column "#Distinct" refers to the number of distinct evaluation points. S is the number of non-zero entries in each matrix in R1CS and HPR. C_m and C_a are the numbers of multiplication and addition gates, respectively. The matrices in R1CS and HPR are of size $C_m \times C_m$. The "*" refers to the verifier-efficient version, i.e., the vector combination technique is applied.

PIOP	#Polys	#Evals	#Distinct	Max Degree	Vector Size
VOR1CS	10/4	4	2	$2C_m + 3S$	$3S$
VOR1CS*	7/4	3	2	$C_m + 6S$	$C_m + 3S$
VOHPR	10/4	4	2	$2C_m + 3S$	$3S$
VOHPR*	7/4	3	2	$5C_m + 6S$	$3C_m + 3S$
VOPLONK	6/2	3	2	$5C_m + 4C_a$	$3C_m + 3C_a$
VOPLONK*	5/2	3	2	$5C_m + 6C_a$	$3C_m + 3C_a$
Marlin	9/12	18	3	$6S + 6$	-
PLONK	8/6	7	2	$9C_m + 9C_a$	-

5.2 Comparison of Proof Size

We compare our zkSNARKs to state-of-the-art constructions with constant verification time and proof size: Marlin [16], Groth16 [23] and PLONK [21], where Groth16 requires a per-circuit trusted setup, while others (including ours) support universal setups for all circuits within a certain bound. Although Sonic [25] also belongs to this category of zkSNARKs, we do not include it in comparison because there is no working implementation of the unhelped version of Sonic at the time of writing. There are also zkSNARKs that support transparent setup, i.e., the setup can be executed by an untrusted party. However, these zkSNARKs typically have much larger proof sizes and verification costs than the ones relying on trusted setups, so we also exclude them from the comparison.

The implementations of Marlin, Groth16 and PLONK we select are all based on the `arkworks` toolchain. Specifically, the Marlin and Groth16 implementations are developed by the `arkworks` team. This team has not provided an implementation of PLONK at the time of writing, so we picked another implementation[11] that uses the same toolchain. The number for Marlin is reported in the document of the `arkworks` implementation of Marlin, which is smaller than the number reported in the Marlin paper [16]. For other zkSNARKs, we use the `ark-serialize` crate for encoding the proof and measuring the proof size. The number for PLONK is larger than we expected for reasons that will be discussed later.

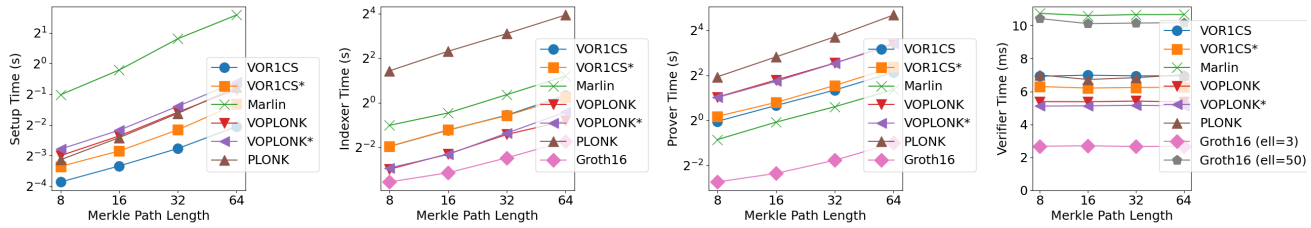
Table 2 shows that all the VOProof-based zkSNARKs have shorter proofs than Marlin and PLONK.[12]

5.3 Comparison of Running Time

To evaluate the running time of the zkSNARKs, we execute the setup, the indexing, and the proving algorithms on an HP-Z8-G4 workstation with the system Ubuntu 20.04 LTS, 512 GB memory,

[11]https://github.com/ZK-Garage/plonk

[12]Although the proof size of Groth16 is significantly smaller than others, this performance comes at the cost of per-circuit trusted setup.



(a) Setup Time (b) Indexing Time (c) Proving Time (d) Verification Time

Figure 6: Comparison between the running time of zkSNARKs. The "*" refers to the verifier-efficient versions. The Groth16 verification time depends linearly on the public input size (denoted by ell). Note that the lines for VOPLONK and VOPLONK* are very close in the left three subfigures.

Table 2: Comparison of Proof Sizes (in Bytes), with BLS12-381 Curve. The "*" refers to the verifier-efficient version.

zkSNARK	Proof Size	zkSNARK	Proof Size	zkSNARK	Proof Size
VOR1CS*	496	VOHPR*	496	VOPLONK*	400
VOR1CS	672	VOHPR	672	VOPLONK	448
Marlin	784	Groth16	192	PLONK	1040

and 2.30 GHz 64-core Intel(R) Xeon(R) Gold 5218 CPU. For the verification algorithm, we run it on a laptop with a MacOS system, 8 GB memory, and a 3.1 GHz Intel Core i5 CPU. We choose different platforms because in real-life situations the zkSNARK proof generation is time-consuming and is expected to run on a powerful machine, while the proof verification is typically running on machines with weaker computation power, e.g., a laptop. The programs are executed in single threads.

We select the Merkle-path verification as the benchmarking computation, which is a common component in various applications. The hash function we choose for the Merkle-tree is Poseidon [22], a SNARK-friendly cryptographic hash function. We exclude VOHPR from the evaluation, due to the lack of tools for transforming circuits into HPR matrices.

Figure 6 shows that VOProof-based zkSNARKs outperform PLONK and Marlin in setup, indexing and verification.

Setup time. Groth16 does not have the universal setup step and is not included in the comparison of this metric. The setup time depends on the maximal polynomial degree and the polynomial commitment scheme. VOPLONK and its verifier-efficient variant have roughly the same performance compared to PLONK, which is actually unexpected, because VOPLONK has a larger degree for the same circuit. The reasons for VOPLONK performing the same as PLONK despite the larger maximal degree include: a) VOPLONK uses the original polynomial commitment scheme proposed in the PLONK paper [21] which has fast setup, while the implementation of PLONK in our experiment uses the KZG10 scheme provided in the `arkworks` toolchain, which is heavier than the one in [21] (this also explains the larger-than-expected proof size); and b) PLONK requires the degree to be padded to a power of two, while VOPLONK does not. In comparison, both versions of VOR1CS have a shorter setup time than the rest.

Indexing time. Both VOR1CS and VOPLONK perform better than the state of the art, except Groth16.

Proving time. Both VOR1CS and VOPLONK are outperformed by Marlin, and Groth16 still has the best performance.

Verification time. VOR1CS and VOPLONK are competitive compared to others. Note that the verification time of Groth16 depends heavily on the size of public inputs to the circuit. Specifically, the verification time is dominated by two pairings (three for Groth16) and scalar-multiplications in $\mathbb{G}_1$. For Groth16, each public input contributes one more scalar-multiplication. For the other zkSNARKs, the size of public inputs only affects the number of field operations which are considerably cheaper than group operations. Therefore, we benchmarked Groth16 with 3 and 50 public inputs respectively. Figure 6d shows that the verifier of Marlin is roughly the same as Groth16 when the number of public inputs is 50. In comparison, VOR1CS, VOPLONK and PLONK have a smaller number of scalar-multiplications (roughly 20 as we estimated).

Remarks. Most of the efficiency improvements of the VOProof-based zkSNARKs result from combining the monomial-basis techniques from Claymore and several techniques from PLONK [21], including the lightweight KZG and exploitation of the homomorphic addition of polynomial commitments. Although these optimizations are not VO-specific, i.e., we can present all our constructions and apply these optimizations without the VO formalization, such presentations are cumbersome and repetitive: the compilation logic must be adapted and repeated in every protocol's description.

Note that the implementation of PLONK in our experiment could benefit from choosing the lightweight polynomial commitment presented in [21] and perform better than as shown in Table 2 and Fig. 6. However, Marlin may not switch to this lightweight scheme because Marlin requires individual degree bound for the committed polynomials, while PLONK and all the VOProof-based zkSNARKs do not have this requirement.

6 CONCLUSION

We introduced VOProof, which simplifies designing zkSNARKs for various specialized problems and constraint systems for general-purpose computations. Although sufferring from the inevitable loss of generality and flexibility compared to directly using PIOP, our new workflow enjoys the benefit of simplicity. Such simplicity comes from: (1) on the application side, VO provides an interface

matching with the operations of constraint systems, and (2) on the implementation side, VO conceals the application-specific logics to focus on implementing the VO queries. Such functional separation opens the possibility for a broad class of zkSNARKs with different combinations of properties.

The zkSNARKs generated using our tool show competitive efficiency compared to the state of the art. In particular, they outperform prior works in proof sizes and verification times, with the sole exception of Groth16 whose efficiency comes at the cost of a circuit-specific trusted setup. We believe such advantage in verification efficiency outweighs the slight sacrifice in proving efficiency compared to Marlin and Groth16. For example, on Ethereum, assuming an ETH is $2000, the transaction fee for verifying each SNARK proof is typically two magnitudes greater than the cost of generating this proof [2].

Future work. The VOProof methodology reveals many opportunities for further investigation. The zkSNARKs in this work might be improved by an alternative compiler, e.g., a compiler working in the Reed-Solomon basis, or a compiler that circumvents PIOP and compiles VO protocols directly into zkSNARKs. Another promising direction is to unify all prior zkSNARKs in one framework using the language of the VO model. Analyzing existing protocols in this framework may provide more thorough explanations for the different features in prior constructions, and potentially reveal new directions for improvements.

Acknowledgement. We thank the anonymous reviewers for their constructive comments. This work is partially supported by National Key Research and Development Project 2020YFA0712300. This work is partially supported by Blockchain Research Grants of Cryptape. Ren is partially supported by Shandong Key Research and Development Program (Grant No. 2020ZLYS09).

REFERENCES

[1] 2021. Aztec. (2021). https://zk.money.
[2] 2021. ZK-Rollup development experience sharing. (2021). https://www.fluidex.io/en/blog/zkrollup-intro1/.
[3] 2021. zkSync. (2021). https://zksync.io.
[4] Carsten Baum, Jonathan Bootle, Andrea Cerulli, Rafaël del Pino, Jens Groth, and Vadim Lyubashevsky. 2018. Sub-linear Lattice-Based Zero-Knowledge Arguments for Arithmetic Circuits. In *Advances in Cryptology - CRYPTO 2018 (LNCS)*, Vol. 10992. Springer, 669–699. https://doi.org/10.1007/978-3-319-96881-0_23
[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.* 2018 (2018), 46. http://eprint.iacr.org/2018/046
[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014*. IEEE Computer Society, 459–474. https://doi.org/10.1109/SP.2014.36
[7] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology - EUROCRYPT 2019 (LNCS)*, Vol. 11476. Springer, 103–128. https://doi.org/10.1007/978-3-030-17653-2_4
[8] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Innovations in Theoretical Computer Science 2012*. ACM, 326–349. https://doi.org/10.1145/2090236.2090263
[9] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. ACM, 103–112. https://doi.org/10.1145/62212.62222
[10] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 2016. Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting. In *Advances in Cryptology - EUROCRYPT 2016 (LNCS)*, Vol. 9666. Springer, 327–357. https://doi.org/10.1007/978-3-662-49896-5_12
[11] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K. Jakobsen. 2017. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 336–365.
[12] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 315–334. https://doi.org/10.1109/SP.2018.00020
[13] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. 2020. Transparent SNARKs from DARK Compilers. In *Advances in Cryptology - EUROCRYPT 2020 (LNCS)*, Vol. 12105. Springer, 677–706. https://doi.org/10.1007/978-3-030-45721-1_24
[14] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. 2021. Lunar: A Toolbox for More Efficient Universal and Updatable zkSNARKs and Commit-and-Prove Extensions. In *Advances in Cryptology - ASIACRYPT 2021, Proceedings, Part III (Lecture Notes in Computer Science)*, Mehdi Tibouchi and Huaxiong Wang (Eds.), Vol. 13092. Springer, 3–33. https://doi.org/10.1007/978-3-030-92078-4_1
[15] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2075–2092. https://doi.org/10.1145/3319535.3339820
[16] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology - EUROCRYPT 2020 (LNCS)*, Vol. 12105. Springer, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26
[17] Alessandro Chiesa, Fermi Ma, Nicholas Spooner, and Mark Zhandry. 2021. Post-Quantum Succinct Arguments: Breaking the Quantum Rewinding Barrier. Cryptology ePrint Archive, Report 2021/334. (2021). https://ia.cr/2021/334.
[18] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. 2020. Fractal: Post-quantum and Transparent Recursive Proofs from Holography. In *Advances in Cryptology - EUROCRYPT 2020 (LNCS)*, Vol. 12105. Springer, 769–793. https://doi.org/10.1007/978-3-030-45721-1_27
[19] Amos Fiat and Adi Shamir. 1986. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology - CRYPTO '86 (LNCS)*, Vol. 263. Springer, 186–194. https://doi.org/10.1007/3-540-47721-7_12 https://doi.org/10.1007/3-540-47721-7_12.
[20] Ariel Gabizon and Zachary J. Williamson. 2020. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.* 2020 (2020), 315. https://eprint.iacr.org/2020/315
[21] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *IACR Cryptol. ePrint Arch.* 2019 (2019), 953. https://eprint.iacr.org/2019/953
[22] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 519–535.
[23] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology - EUROCRYPT 2016 (LNCS)*, Vol. 9666. Springer, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
[24] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology - ASIACRYPT 2010 (LNCS)*, Vol. 6477. Springer, 177–194. https://doi.org/10.1007/978-3-642-17373-8_11
[25] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings. *IACR Cryptol. ePrint Arch.* 2019 (2019), 99. https://eprint.iacr.org/2019/099
[26] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013*. IEEE Computer Society, 238–252. https://doi.org/10.1109/SP.2013.47
[27] Carla Ràfols and Arantxa Zapico. 2021. An algebraic framework for universal and updatable SNARKs. In *Annual International Cryptology Conference*. Springer, 774–804.
[28] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology - CRYPTO 2020 (LNCS)*, Vol. 12172. Springer, 704–737. https://doi.org/10.1007/978-3-030-56877-1_25
[29] Alan Szepieniec and Yuncong Zhang. 2022. Polynomial IOPs for Linear Algebra Relations. In *Public-Key Cryptography - PKC 2022, Proceedings, Part I*, Vol. 13177. Springer, 523–552. https://doi.org/10.1007/978-3-030-97121-2_19
[30] Justin Thaler. 2020. Proofs, arguments, and zero-knowledge. (2020).
[31] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology - CRYPTO 2019 (LNCS)*, Vol. 11694. Springer, 733–764. https://doi.org/10.1007/978-3-030-26954-8_24

[32] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM with Program-Independent Preprocessing. In *2018 IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 908–925. https://doi.org/10.1109/SP.2018.00013

A ALTERNATIVE PROTOCOLS

We present in Fig. 7 the verifier-efficient versions of the SMVP and VOPLONK protocols where the vector combination technique is applied. The offline protocol of SMVP is the same as in Fig. 5 and is omitted here.

R1CS-lite [14], defined in Equation (13), is a variant of R1CS that is more lightweight yet still NP-complete. It is straightforward to adapt VOR1CS for R1CS-lite, and the VO protocol is presented in Fig. 8.

$$\mathcal{R}_{\mathsf{R1CSLite}} = \left\{ \left(\begin{pmatrix} H, \ell \\ \boldsymbol{A}, \boldsymbol{B} \end{pmatrix}, \begin{matrix} \boldsymbol{x}, \\ \boldsymbol{w} \end{matrix} \right) \middle| \begin{matrix} \boldsymbol{A}, \boldsymbol{B} \in \mathbb{F}^{H \times H} \\ \boldsymbol{x} \in \mathbb{F}^{\ell}, \boldsymbol{w} \in \mathbb{F}^{H-\ell-1} \\ (\boldsymbol{A}\boldsymbol{z}) \circ (\boldsymbol{B}\boldsymbol{z}) = \boldsymbol{z} \\ \text{where } \boldsymbol{z} = 1 \| \boldsymbol{x} \| \boldsymbol{w} \end{matrix} \right\} \tag{13}$$

procedure $\mathsf{SMVP}_{\mathsf{online}}(h_{\boldsymbol{a}}, h_{\boldsymbol{b}}; \boldsymbol{a}, \boldsymbol{b})$

V samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends α to P;

P computes $\boldsymbol{r}_\alpha := \left(\frac{1}{\alpha - \gamma^i}\right)_{i=1}^{H}$ and $\boldsymbol{c} := \boldsymbol{r}_\alpha^{\mathsf{T}} \boldsymbol{M}$;

P submits $\boldsymbol{s} := \boldsymbol{r}_\alpha \| \boldsymbol{c}$ and sends $h_{\boldsymbol{s}}$ to V;

V checks $\boldsymbol{s} \circ (\alpha \cdot \boldsymbol{1}^H - \boldsymbol{\gamma}^H) \stackrel{?}{=}_n \boldsymbol{1}^H$;

V checks $\langle \boldsymbol{s}, \boldsymbol{b} \| (-\boldsymbol{a}) \rangle \stackrel{?}{=} 0$ and $\boldsymbol{s} \circ \left(\boldsymbol{1}^{n-(H+K)}\right)^{\to H+K} \stackrel{?}{=}_n \boldsymbol{0}$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends β to P;

P computes $\boldsymbol{r}_\beta := \left(\frac{1}{\beta - \gamma^i}\right)_{i=1}^{K}$ and $\boldsymbol{t} := \left(\frac{1}{(\alpha - \gamma^{\mathrm{row}_i})(\beta - \gamma^{\mathrm{col}_i})}\right)_{i=1}^{S}$;

P submits $\boldsymbol{h} := \boldsymbol{r}_\beta \| \boldsymbol{t}$ and sends $h_{\boldsymbol{h}}$ to V;

V checks $\boldsymbol{h} \circ (\beta \cdot \boldsymbol{1}^K - \boldsymbol{\gamma}^K) \stackrel{?}{=}_n \boldsymbol{1}^K$;

V checks $\boldsymbol{h} \circ (\alpha\beta \cdot (\boldsymbol{0}^K \| \boldsymbol{1}^{n-K}) - \alpha \cdot \boldsymbol{w}^{\to K} - \beta \cdot \boldsymbol{u}^{\to K} + \boldsymbol{y}^{\to K}) \stackrel{?}{=}_n \boldsymbol{0}^K \| \boldsymbol{1}^S$;

V checks $\langle \boldsymbol{h}^{\to H}, \boldsymbol{c} \rangle \stackrel{?}{=} \langle \boldsymbol{t}, \boldsymbol{v}^{\to K} \rangle$.

procedure $\mathsf{VOPLONK}_{\mathsf{offline}}(C_c, C_a, C_m, \boldsymbol{d}, \sigma, \mathcal{I}_x)$

Let $C = C_a + C_m$;

Let $\tau \in \Sigma([3C + C_c])$ be a permutation that swaps $[C+1..2C]$ and $[2C+1..3C]$, i.e., for $i \in [C]$, $\tau(C+i) = 2C+i$ and $\tau(2C+i) = C+i$;

Let $\pi \in \Sigma([3C + C_c])$ swap $[2C+1..2C+C_m]$ and $[2C+C_m+1..3C]$, i.e., for $i \in [C_m]$, $\pi(2C+i) = 2C+C_a+i$ and for $i \in [C_a]$, $\pi(2C+i) = 2C - C_m + i$;

Let $\sigma' := \sigma(\pi(\tau(\cdot)))$;

I submits $\boldsymbol{d}, \boldsymbol{\sigma} := \left(\gamma^{\sigma'(i)-1}\right)_{i=1}^{3C+C_c}$;

I sends $\boldsymbol{d}, \boldsymbol{\sigma}$ to P, and $h_{\boldsymbol{d}}, h_{\boldsymbol{\sigma}}$ to V.

procedure $\mathsf{VOPLONK}_{\mathsf{online}}(\boldsymbol{x}; \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$

P submits $\boldsymbol{u} := \boldsymbol{a} \| \boldsymbol{c} \| \boldsymbol{b}_{[C_m+1..C]} \| \boldsymbol{b}_{[1..C_m]}$ and sends $h_{\boldsymbol{u}}$ to V;

V checks $\boldsymbol{u}^{\to 2C+C_a} \circ \boldsymbol{u} \stackrel{?}{=}_n \boldsymbol{u}^{\to C+C_a} \circ \boldsymbol{1}_{[2C+C_a..3C]}$;

V checks $(\boldsymbol{u}^{\to C+C_a} + \boldsymbol{u}^{\to C_a} - \boldsymbol{u}) \circ \boldsymbol{1}_{[2C+1..2C+C_a]} \stackrel{?}{=}_n \boldsymbol{0}$;

Run $\mathsf{VBatch}(h_{\boldsymbol{u}}, \{\pi(\tau(i)), \boldsymbol{x}_{[i]}\}_{i \in \mathcal{I}_x}; \boldsymbol{u})$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends β to P;

V submits $\boldsymbol{w} := \boldsymbol{u} \| \boldsymbol{d}$, $\boldsymbol{s} := \boldsymbol{\sigma} + \beta \cdot \boldsymbol{w}$, $\boldsymbol{t} := \boldsymbol{\gamma}^{3C+C_c} + \beta \cdot \boldsymbol{w}$;

Run $\mathsf{PermCheck}(h_{\boldsymbol{t}}, h_{\boldsymbol{s}}, 3C + C_c; \boldsymbol{t}, \boldsymbol{s})$.

Figure 7: Verifier-Efficient Versions of VO Protocols.

procedure $\mathsf{VOR1CSLite}_{\mathsf{offline}}(\boldsymbol{A}, \boldsymbol{B})$

Run $\mathsf{SMVP}_{\mathsf{offline}}(\boldsymbol{M} = (\boldsymbol{A}^{\mathsf{T}} \| \boldsymbol{B}^{\mathsf{T}})^{\mathsf{T}} \in \mathbb{F}^{2H \times H})$.

procedure $\mathsf{VOR1CSLite}_{\mathsf{online}}(\boldsymbol{x}; \boldsymbol{w})$

P submits $\boldsymbol{w}$ and $\boldsymbol{y} := \boldsymbol{M}(1 \| \boldsymbol{x} \| \boldsymbol{w})$ and sends $h_{\boldsymbol{w}}, h_{\boldsymbol{y}}$ to V;

Run $\mathsf{SMVP}_{\mathsf{online}}(h_{1\|\boldsymbol{x}\|\boldsymbol{w}}, h_{\boldsymbol{y}}; 1 \| \boldsymbol{x} \| \boldsymbol{w}, \boldsymbol{y})$;

V checks $\boldsymbol{y} \circ \boldsymbol{y}^{\to H} \stackrel{?}{=}_n \boldsymbol{0}^H \| 1 \| \boldsymbol{x} \| \boldsymbol{w}$.

Figure 8: VO Protocol for R1CS-lite.