SoK: Lookup Table Arguments

Hossein Hafezi[1⋆], Gaspard Anthoine[2,4], Matteo Campanelli[3], and Dario Fiore[2]

[1] New York University, USA
[2] IMDEA Software Institute, Spain
[3] Offchain Labs, USA
[4] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. Lookup arguments have become a central tool in proof systems, powering a range of practical applications. They enable the efficient enforcement of non-native operations, such as bit decomposition, range checks, comparisons, and floating-point arithmetic. They underpin zk-VMs by modelling instruction tables, provide set membership proofs in stateful computations, and strengthen extractors by ensuring witnesses belong to small domains. Despite these broad uses, existing lookup constructions vary widely in assumptions, efficiency, and composability. In this work, we systematize the design of lookup arguments and the cryptographic primitives they rely on. We introduce a unified and modular framework that covers standard, projective, indexed, vector, and decomposable lookups. We classify existing protocols by proof technique—multiset equality, Logup-based, accumulators, and subvector extraction (matrix–vector)—as well as by composition style. We survey and evaluate existing protocols along dimensions such as prover cost, dependence on table size, and compatibility with recursive proofs. From this analysis, we distill lessons and guidelines for choosing lookup constructions in practice and highlight the benefits and limitations of emerging directions in literature, such as preprocessing and decomposability.

⋆ Work primarily conducted at IMDEA Software.

Table of Contents

1 Introduction

Succinct non-interactive arguments of knowledge (aka SNARKs) allow a prover to convince a verifier that a statement is valid with a short and efficiently verifiable proof. A striking property of modern SNARKs is that they are general-purpose, namely, they can prove the correctness of arbitrary computation while maintaining succinct proofs and fast verification. This combination has driven widespread adoption, both in theory and practice.

A recent development in this space is the growing interest in *lookup arguments* [Boo+18; GW20; Zap+22a; EFG22; STW24]. Informally, a lookup argument allows proving that each element of a witness vector belongs to a predefined table.[5] Despite their apparent simplicity, this functionality has proven to be a remarkably powerful tool. Notably, it allows the designers of SNARKs to reduce non-native operations–such as bit decomposition, range checks, comparisons, and floating-point arithmetic–to table checks, thereby greatly reducing the number of circuit constraints and prover time. Lookup arguments have already found diverse applications within SNARKs, including efficient proofs for hash functions [Sze+23; Gra+22], modular arithmetic, membership proofs, and general virtual machine executions [AST24].

Research on lookup arguments has progressed at an extraordinary pace. In just a few years, numerous constructions have been proposed, each with distinct design methodologies, efficiency trade-offs, and compatibility requirements. In this rapid progress, the focus was especially on introducing new techniques that improve efficiency. However, this has been often achieved in specialized contexts, using definitions of the lookup functionality tailored to the application or proof system at hand, rather than guided by a general framework. For example, several works [GW20; Zap+22a; EFG22; Zap+22b; Cam+24] develop lookup arguments for witnesses and tables committed using the KZG polynomial commitment [KZG10], implicitly adopting a notion of lookup specialized to this (univariate) polynomial-based setting. The context in which the lookup argument operates is a crucial aspect of its use. Seen as standalone proof systems, lookup arguments may indeed be rather uninteresting as they would only allow proving membership of elements in a table. Their power instead arises when they are composed with other proofs about the same witness. Understanding how, when, and under which assumptions this composition is possible is, therefore, a key question we address.

1.1 Contributions

We provide a systematization of knowledge (SoK) on lookup arguments. We aim to clarify definitions, highlight key methodologies for composing lookup proofs with other proof systems, identify useful variants, analyze the current state of the art, and applications. In doing so, we shed light on the unifying principles underlying existing proposals and guide researchers and practitioners seeking to design or deploy lookup arguments. Our main contributions include:

Definitions. We introduce a modular framework of definitions for lookup arguments. We identify the lookup relation—checking that each $w_j \in \mathbf{t}$ for two vectors $\mathbf{w}, \mathbf{t}$—as the fundamental problem. Then we formalize proof systems for this relation in two flavours: commit-and-prove SNARKs [CFQ19] and polynomial interactive oracle proofs (PIOP) [Chi+20; BFS20]. The former formalizes the case of a (succinct) proof about *committed* vectors $\mathbf{t}$ and $\mathbf{w}$ in which the commitment to $\mathbf{t}$ is usually publicly computed by the verifier in an offline preprocessing phase. This commit-and-prove flavour essentially corresponds to the notion of lookup arguments. The latter formalizes proof systems for the lookup relation in the information-theoretic PIOP framework, in which the vectors $\mathbf{t}$ and $\mathbf{w}$ are encoded as polynomials and available to the verifier as *oracles*. The oracle model is an ideal assumption that can be later removed via a popular compilation strategy that uses polynomial commitments. Formalizing lookup proof systems using this perspective highlights two distinct composition methodologies for using lookups when

[5] Although the term *arguments* refers to computationally sound proof systems, many works in the literature broadly refer to lookup arguments as to the proof techniques used to build argument systems that involve lookup checks.

constructing zkSNARKs: (i) argument-level composition via the commit-and-prove paradigm, and (ii) PIOP-level composition.

Beyond simple lookups. We present several variants of the lookup problem, which include: *indexed lookups*, where the membership relation is enforced with respect to a given vector of indices $\boldsymbol{i}$, i.e., for every j, $w_j = t_{i_j}$; *vector lookups*, where the entries of the witness and table are tuples, and *projective lookups*, where only a subset of the witness entries are checked to be in the table, i.e., for a given set of indices $\boldsymbol{i}$, it holds $w_j \in \mathbf{t}$ for every $j \in \boldsymbol{i}$. We also define the notion of *online lookup tables*, in which the table depends on the witness and hence cannot be preprocessed. We find the projective setting to be an overlooked aspect of lookup tables in prior work. As noted earlier, lookup tables become most useful when composed with other proof systems, where they are typically used to prove that *part of the witness* lies in a table. However, the syntax of lookup relations and arguments in most existing definitions does not capture projectiveness; instead, they enforce that the entire witness must appear in the table. We generalize our definitions to the projective setting and demonstrate that existing techniques—such as the logup lemma (and works building on it, including cq [EFG22] and Protostar [BC23]), as well as the matrix–vector technique (e.g., Lasso [STW24])—naturally extend to the projective setting with minimal overhead.

State of the art and applications. We survey the state of the art in lookup arguments, focusing on three main aspects. First, we distill and present the main algebraic techniques underlying existing lookup arguments, and we map them to the existing schemes (see Section 4). The main categories include multiset equality, accumulators, Logup-based, and subvector extraction (matrix-vector). Second, in Section 5, we examine two practical aspects of lookup arguments: (i) their compatibility, meaning which SNARKs they can be efficiently integrated with, and (ii) their efficiency, in particular how performance scales with the size of the lookup table. When it comes to supporting large tables (i.e., $|\mathbf{t}| \gg \mathbf{w}$), our analysis points out limitations in the state of the art, which either assumes the tables to have some structure (e.g., they can be decomposed in the product of smaller tables or be described by a short function) or relies on preprocessing methods that are concretely too expensive to scale (see Figure 1 and discussion in Appendix A). Third, we provide an overview of the most important applications of lookup arguments, identifying which features matter most in different applications. We group these applications in three main categories: replacement for non-native or non-arithmetic operations, set membership, and verifiable memory.

Through this systematization, our work aims to provide a unified understanding of lookup arguments, distilling lessons and guidelines for choosing lookup constructions in practice and highlighting the benefits and limitations of emerging directions in literature, such as preprocessing and decomposability.

2 Preliminaries

Notation. Let $\mathbb{F}$ be a finite field and $\mathbb{G}$ a group with scalars in $\mathbb{F}$, with additive notation. For $a \in \mathbb{F}$ and $g \in \mathbb{G}$, the scalar multiplication of g by a is denoted as $a \times g$. For an asymmetric pairing group, we define it as a tuple $(p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, where p is the order of groups $\mathbb{G}_1$ and $\mathbb{G}_2$, and e is an efficiently computable, non-degenerate bilinear map. For a natural number n, $[n]$ denotes the set of integers $\{0, 1, \ldots, n-1\}$.

Given a vector $\mathbf{w}$, we denote its associated oracle by $[[\mathbf{w}]]$. The double-bracket notation indicates an *oracle handle* to the underlying data. While this notation can represent various interpretations (depending on context), we do not explicitly distinguish between them for the sake of clarity. For example, $\mathbf{w}$ might represent the evaluations of a multilinear polynomial over the Boolean hypercube. In this case, the oracle $[[\mathbf{w}]]$ allows the client to query the corresponding multilinear polynomial at any point in the domain. Let $\mathbf{t} \in \mathbb{S}^N$ denote the lookup table, represented as a vector of length N, and let $\mathbf{w} \in \mathbb{S}^n$ denote the witness, represented as a vector

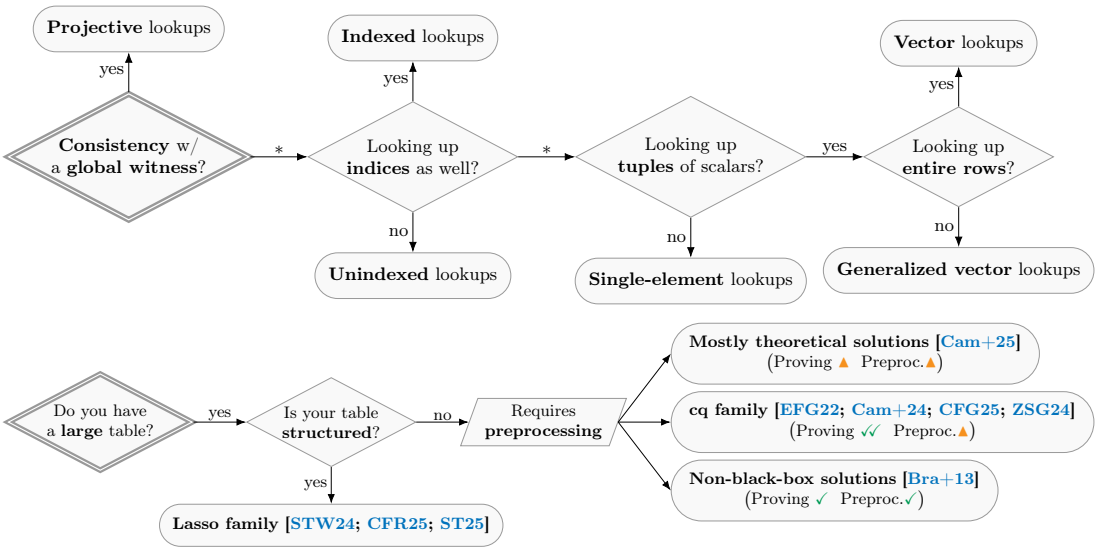

Fig. 1: *Above*: mapping requirements on lookup relations to properties in respective schemes. *Below*: an informal performance taxonomy of schemes for large tables.

of length n. We use the notation $M_{n\times m}$ to refer to a matrix with n rows and m columns; $M_{i,j}$ denotes the entry on row i and column j of M.

Polynomial commitment scheme (PCS) [KZG10]. PCS is a cryptographic primitive that enables a prover to *commit* to a polynomial succinctly and later prove the correctness of its evaluation at some chosen points from the domain, without revealing the full polynomial. When a verifier requests an opening at a point x, the prover returns both the claimed evaluation $y = p(x)$ and a proof π attesting to its correctness. The verifier can efficiently check that y is the correct value of the committed polynomial at x using the proof π. For a formal definition, we refer the reader to Definition 10. Here, the polynomial can have different types, such as univariate, multilinear or multivariate.

Sumcheck protocol [Lun+92]. Let g be some ℓ-variate polynomial defined over a finite field $\mathbb{F}$. The purpose of the sumcheck protocol is for the prover to provide the verifier with the following sum: $H := \sum_{\mathbf{b}\in\{0,1\}^\ell} g(\mathbf{b})$. To compute H unaided, the verifier would have to evaluate g at all 2^ℓ points in $\{0,1\}^\ell$ and sum the results. The sumcheck protocol is an interactive protocol that allows the prover to reduce a claim on summation above to a claim on an evaluation of g at a random point. The verifier's runtime is $O(\ell)$, plus the time required to evaluate g at a single point $\mathbf{r} \in \mathbb{F}^\ell$, this evaluation can be provided by the prover directly by committing to g and opening it at point $\mathbf{r}$. This is exponentially faster than the 2^ℓ time that would generally be required for the verifier to compute H. There is a similar lemma known as *univariate lemma* (Lemma 1), which is popular in the context of univariate polynomials.

Argument of knowledge. An argument of knowledge (called an SNARK if succinct) lets a prover prove to a verifier that it knows a witness to an NP statement. It can support properties such as *succinctness*, i.e. the verifier's computation being sublinear in the witness size or *zero-knowledge*, i.e. the verifier learns nothing beyond the satisfiability of the statement. For a formal definition, see Definition 11 in Appendix B.

Accumulation scheme [BC23]. An accumulation scheme is a fundamental building block for recursive proof systems. At a high level, it enables a prover and a verifier to reduce the satisfiability of two NP statements into a single new statement through a lightweight interaction between the two parties. For a formal definition, we refer the reader to Definition 12 in the appendix. We expand more on recursive proofs and accumulation schemes in Appendix H.

Commit-and-prove SNARKs [CFQ19]. The commit-and-prove paradigm enables one to prove *commit-and-prove relations*. In such relations, the instance contains a commitment to part of the witness, even potentially the entire witness. This feature is particularly useful for composing proof systems. For example, when composing a lookup argument with another proof system, the instance of the proof system contains a commitment to the relevant portion of the witness (which may even be the entire witness) that will be checked against the lookup table. Since lookup arguments are naturally commit-and-prove (their instances also include commitments to the looked-up values), this shared commitment structure allows for seamless composition between the two arguments.

Multilinear extensions. An n-variate polynomial $p : \mathbb{F}^n \to \mathbb{F}$ is said to be *multilinear* if p has degree at most one in each variable. Let $f : \{0,1\}^n \to \mathbb{F}$ be any function mapping the n-dimensional Boolean hypercube to a field $\mathbb{F}$. Then there exists a unique multilinear polynomial $\tilde{f} : \mathbb{F}^n \to \mathbb{F}$ that extends f to $\mathbb{F}^n$. This is referred to as the *multilinear extension (MLE)* of f. We will denote by $\widetilde{\mathsf{eq}}$ the MLE of the function $\mathsf{eq} : \{0,1\}^s \times \{0,1\}^s \to \{0,1\}$ such that $\mathsf{eq}(x,e) = 1$ if and only if $x = e$. An explicit expression for $\widetilde{\mathsf{eq}}$ is:

$$\widetilde{\mathsf{eq}}(x,e) = \prod_{i=1}^{s} \big(x_i e_i + (1-x_i)(1-e_i)\big).$$

Multilinear extensions of vectors. Given a vector $u \in \mathbb{F}^m$, we denote its multilinear polynomial by $\tilde{u}$ (obtained by viewing u as a function mapping $\{0,1\}^{\log m} \to \mathbb{F}$ in the natural way). Later in Appendix G.3, when we explain Spark, we define a multilinear extension of a matrix following the notation of Spartan [Set20]. A matrix $M_{n\times m}$ can also be seen as a function $M : \{0,1\}^{\log n} \times \{0,1\}^{\log m} \to \mathbb{F}$; its multilinear extension is defined as a natural extension of the vector case.

3 A Unified Definitional Framework for Lookups

In this section, we present a modular framework for defining *lookup arguments*. Unlike most of the prior work—which typically defines the lookup argument directly or encodes the lookup witness and table as evaluations of a univariate polynomial (e.g., using KZG [KZG10])—our approach is modular. This modularity facilitates seamless composition of the lookup argument with existing proof systems. At its core, in Section 3.1, we define a *lookup relation* (Definition 1)—a polynomial-time relation that asserts whether a given vector appears within a fixed table. We begin by formalizing a basic (deterministic) relation for lookup: given a table $\mathbf{t}$ parametrizing the relation, a vector $\mathbf{w}$, passed as public input, is contained in $\mathbf{t}$. The deterministic relation above is not very useful by itself. In fact, the real power of lookup stems from its ability to be composed with other relations. In Section 3.2, we then extend it to its more useful, non-deterministic version where the verifier only has a "handle" to the witness. We introduce this non-deterministic relation in two forms to support different modes of composition with proof systems: the *committed lookup relation* (Definition 2) and the *oracle lookup relation* (Definition 3). In each, the "handle" is instantiated with a commitment and an oracle, respectively. We elaborate on both approaches later in this section.

In Section 3.3, we define *projective lookup*, a generalization of the standard lookup relation. Unlike the standard lookup, where the prover demonstrates that *all* the elements in the witness belong to a table, projective lookups allow the prover to assert that *only certain indices of the witness* appear in the table. This variant aligns better with practical use cases, where a proof system often requires checking membership for specific indices of the witness only (e.g., verifying that certain elements in the witness are small). We next define two further generalizations of the lookup relation: *indexed lookup* and *vector lookup*, presented in Sections 3.4 and 3.5, respectively. In an indexed lookup, both the witness and the table are treated as ordered lists (i.e., vectors), and the order of elements becomes significant. In this setting, the verifier receives a commitment to the relevant subtable along with a commitment to the indices of its elements in

the original table. This formulation enables new applications, such as proving custom functions without encoding the function logic directly into the arithmetic circuit. We then define *vector lookup*, a natural extension where each entry in the table is a tuple of elements rather than a single element. Next, in Section 3.6, we define the notation of *an online lookup table*, such that the table itself is part of the witness, instead of being in the index. Finally, in Appendix C, we define the notion of *decomposability*, a property of lookup tables that enables the prover to reduce lookups in a large table to lookups in a number of smaller tables, improving the prover's efficiency. This is particularly useful for constructing lookups over very large tables (e.g., size 2^{128}), which cannot be materialized explicitly.

3.1 Lookup relation

Definition 1 (Lookup Relation). *Given a finite set $\mathbb{S}$, a lookup index is defined as $\mathrm{I} := (\mathbb{S}, n, \mathbf{t})$ where n is the number of lookups and $\mathbf{t} \in \mathbb{S}^N$ is the table, the relation LK_I is the set of tuples $\mathbf{w} \in \mathbb{S}^n$ such that $w_i \in \mathbf{t}$ for all $i \in [n]$. In other words:*

$$\mathrm{LK}_\mathrm{I} = \{\mathbf{w} \in \mathbb{S}^n \ : \ \mathbf{w} \subseteq \mathbf{t}\}$$

Note that an index is valid if $0 < n$, N and $\mathbf{t} \in \mathbb{S}^N$. We slightly abuse notation by extending the subset relation $A \subseteq B$ to apply to multisets as well.

What is the finite set $\mathbb{S}$? The lookup relations introduced above are defined generically over a finite set $\mathbb{S}$. In practice, different lookup constructions instantiate $\mathbb{S}$ with specific finite structures—most commonly a finite field—to exploit certain algebraic properties. Early constructions, particularly those based on the KZG commitment scheme, typically operated over large prime fields of 256 bits [GW20; GK22; Zap+22a; PK22; Zap+22b; EFG22]. Other approaches, such as Logup [Hab22] and Lasso [STW24], introduced techniques that extend the applicability of lookup relations to smaller prime fields [Pol22] (e.g., 64-bit fields), binary fields [DP25], and even to rings in recent work [Cry; Gar+25].

Treating table and witness as vectors instead of sets. In the discussion above, we defined the lookup relation using the convention that the table $\mathbf{t}$ is represented as an ordered tuple in $\mathbb{S}^N$, i.e., as a vector. Alternatively, one could define lookup relations over sets rather than vectors. Some constructions [Cam+22a] support only this *set-based* lookup formulation. However, this perspective is generally less compelling for two main reasons. First, the commitment schemes, such as *vector* [CF13] or *polynomial commitment schemes*, employed in SNARKs naturally treat the witness as an ordered list rather than sets. Second, the compiler from general lookup relations to indexed lookup relations is not applicable in the set-based setting, yet indexed lookups are a crucial form used in many SNARK constructions. We also regard the witness $\mathbf{w}$, which is originally a multiset, as an ordered tuple in $\mathbb{S}^n$. As we will see in the next section, the witness must be committed by the prover, while the verifier only receives a commitment to it. Since there is no standard notion of a *multiset commitment scheme*, we instead model the lookup instance as an ordered collection, such as a vector. Consequently, we assume that it is committed using, for instance, a vector commitment scheme or a polynomial commitment scheme.

Duplicates in the table. In the standard definition of a lookup, we can assume without loss of generality that the table $\mathbf{t}$ contains no duplicate entries, as this does not affect the rest of the protocol. However, in the case of indexed lookups (see Definition 7)—where the table is treated as a vector and the index is part of each entry—repetitions in the values are allowed. This is because entries with different indices are considered distinct, even if their values are the same.

3.2 Lookup relation interfaces for composition

The relation LK_I belongs to the class P because it does not involve a witness; it simply consists of an index and an instance $\mathbf{w}$. We now introduce the notion of a *committed lookup relation*, where the instance is a commitment to $\mathbf{w} \in \mathrm{LK}_\mathrm{I}$, and the witness is $\mathbf{w}$ in plaintext.

Definition 2 (Committed Lookup Relation). *Given lookup index* $\mathrm{I} := (\mathbb{S}, n, \mathbf{t})$ *and its corresponding relation* LK_I *and a commitment scheme* $\mathcal{C} = (\mathrm{Com}, \mathrm{Open})$ *on* $\mathbb{S}^n$*,* CLK *is defined as follows:*

$$\mathrm{CLK}_{\mathrm{I},\mathcal{C}} = \{(c;\, \mathbf{w}) \,:\, \mathbf{w} \in \mathrm{LK}_\mathrm{I},\, c = \mathrm{Com}(\mathbf{w})\}$$

Even though a committed lookup relation is a natural notation, it isn't easy to compile it into a PIOP directly. That's why, similar to HyperPlonk [Che+23], we define the following notion of lookup relation where the verifier has oracle access to the witness. In practice, the oracle is typically instantiated using a polynomial commitment scheme. However, treating this commitment as an oracle, enables the composition of the oracle lookup relation with the proof system at the PIOP level. Concretely, one can construct a PIOP for each relation, compose them sequentially, and finally compile the protocol into an argument of knowledge by replacing the oracles with polynomial commitments.

Definition 3 (Oracle Lookup Relation). *Given lookup index* $\mathrm{I} := (\mathbb{S}, n, \mathbf{t})$ *and its corresponding relation* LK_I*,* OLK *is defined as follows:*

$$\mathrm{OLK}_\mathrm{I} = \{([[\mathbf{w}]];\, \mathbf{w}) \,:\, \mathbf{w} \in \mathrm{LK}_\mathrm{I}\}$$

Composition of lookup arguments with proof systems. We define both the committed lookup relation and the oracle lookup relation to support different modes of composability. There are two main ways to compose a lookup with an existing proof system[6]. The first approach is composition at the argument level, leveraging the commit-and-prove paradigm [CFQ19], where both the lookup argument and the underlying proof system share commitments to the same witness. This shared commitment facilitates seamless composition. The second approach operates at the PIOP level, by sequentially composing PIOPs [BSCS16]. More concretely, assuming the existence of a PIOP for the lookup relation and one for the main proof system, one can compose them directly. The composed PIOP can then be compiled into an argument of knowledge by replacing the interactive oracles with polynomial commitments. Each approach suits different contexts. For instance, in Groth16 [Gro16], the argument is constructed directly without relying on a PIOP. To compose it with a lookup argument, one needs a commit-and-prove variant such as Mirage [Kos+20], together with a commitment scheme compatible with the committed lookup relation. In contrast, for systems such as Plonk [GWC19], which first defines a PIOP and then compiles it into an argument using a PCS, we can directly compose the lookup PIOP with the proof system's PIOP and then compile the composed PIOP into an argument of knowledge. Generally, composition via the commit-and-prove paradigm is more general, since it is possible to compile the PIOPs into arguments of knowledge first and then compose them via the commit-and-prove paradigm.

One key difference is that the commit-and-prove paradigm offers better modularity and a more plug-and-play architecture. In this paradigm, the underlying components, namely, the lookup argument and the main proof system, remain separate and only need to share a single commitment. This separation simplifies the developer experience, making it easier to implement and maintain. In contrast, composing components at the PIOP level requires tightly coupling the lookup argument with the proof system. Although this makes development more complex, it can lead to greater efficiency and smaller proof sizes. For example, it enables batching of the lookup polynomial openings with those of the main proof system, reducing overall proof size.

Preprocessing PIOP [Chi+20; BFS20]. A preprocessing PIOP is a type of PIOP which appears frequently in the context of lookup PIOPs. At a high level, this protocol consists of two phases: an *offline phase* and an *online phase*. In the offline phase, an honest indexer, given the index of the relation, generates a set of preprocessed polynomials p_i. These polynomials are provided to the prover and can be queried by the verifier during the online phase. The online phase proceeds similarly to a standard PIOP. The prover, given the instance, the witness, and access to the preprocessed polynomials, interacts with the verifier. The verifier, in turn,

[6] This discussion applies more generally to proof schemes beyond lookups.

has access to the instance and oracle access to the preprocessed polynomials. The goal of this interaction is for the prover to convince the verifier that it possesses a valid witness for the given instance. See Definition 13 for a formal definition of a preprocessing PIOP. The notation of preprocessing PIOPs has been used in SNARKs such as Plonk [GWC19] and Spartan [Set20], as they preprocess the circuit structure and produce some polynomial commitments (e.g., the sparse R1CS matrices A, B, and C in the case of Spartan). Since the polynomials generated by the index are all committed honestly, it was observed in Campanelli et al. [Cam+25] that it suffices for the underlying polynomial commitment scheme to be *weakly binding*. In contrast, the prover-generated polynomials may be generated maliciously, requiring the underlying PCS to be *evaluation binding*. In the context of preprocessing PIOPs for lookups, intuitively, the offline phase can be seen as a preprocessing stage where, given the table $\mathbf{t}$, an honest preprocessor computes some auxiliary polynomials. These polynomials are designed so that, during the online phase, the prover can perform its computation more efficiently.

Lookup argument. A lookup argument is an argument of knowledge for the committed lookup relation. Such an argument can be constructed either directly, for the committed lookup relation, or by first designing a (possibly preprocessing) PIOP for the oracle lookup relation and then compiling it into an argument of knowledge by replacing the oracles with a polynomial commitment scheme.

Efficiency of lookup arguments. Following the terminology of prior work [CFG25], we define a lookup argument as *sublinear* if the prover's runtime is *sublinear* in the size of the lookup table $\mathbf{t}$. If the prover's runtime is *independent* of the size of $\mathbf{t}$, we refer to the argument as *super-sublinear*.[7] Similarly, consider a compound PIOP composed of a lookup PIOP and the main proof system PIOP. When this compound PIOP is compiled into an argument of knowledge, we say that the resulting lookup argument is *sublinear* if the prover runtime is sublinear in the size of $\mathbf{t}$ and *super-sublinear* if the runtime is completely independent of the size of $\mathbf{t}$. We do not define a lookup PIOP itself to be efficient, since the efficiency of the resulting argument also depends on the polynomial commitment scheme used to compile it into an argument. For example, consider cq, which is a generic PIOP that can be compiled with different polynomial commitment schemes beyond KZG. In such cases, the prover's running time is not necessarily independent of the table size. The efficiency of cq specifically arises from its compilation with KZG and the underlying properties of KZG, such as homomorphism.

Zero-knowledge lookup. Zero-knowledge for lookup arguments is defined analogously to standard arguments of knowledge. Specifically, given a lookup instance and index, there exists a polynomial-time simulator that can generate a proof computationally indistinguishable from a real proof. Since the table is considered part of the public index rather than the witness, it is not hidden in the proof; however, there are examples such as Duplex [Han+25] and cq++ [Cam+24], which consider the table as the witness, so it is hidden too. Most lookup schemes are typically presented as a lookup PIOP, which can be compiled into a zero-knowledge argument when the underlying PIOP is zero-knowledge and the polynomial commitment is hiding and supports zero-knowledge opening too. Nevertheless, most existing lookup PIOP schemes are introduced without zero-knowledge, which is left for extension. For example, PIOPs described in cq [EFG22] and Lasso [STW24] are not zero-knowledge, whereas schemes such as Caulk [Zap+22a] and Caulk+ [PK22] do provide zero-knowledge guarantees.

3.3 Projective lookup

Lookup tables are primarily used as subroutines in proof systems to verify that specific components of a witness vector are contained within a predefined table, rather than checking the entire witness. This selective verification is not compatible with the standard lookup definitions previously introduced, such as Definition 1, which require the full witness vector to match

[7] More precisely, we replace the original terms *efficient* and *super-efficient* with *sublinear* and *super-sublinear*, respectively. We later show that there exist tables of practical interest that are not large, and hence the prover's complexity relative to the table size does not become a bottleneck.

a table entry. To bridge this gap, inspired by [Ang+24; CFG25], we introduce the notion of a *projective lookup*. In a projective lookup, given a witness $\mathbf{w} \in \mathbb{S}^m$, we fix a set of indices $\mathbf{i} = (i_1, i_2, \ldots, i_n)$ and check whether the projection $(w_{i_1}, \ldots, w_{i_n})$ belongs to the table $\mathbf{t}$. This concept, referred to as the *projective lookup relation*, allows proof systems to verify properties of partial witnesses—for example, ensuring that some components consist of small field elements—thereby supporting more flexible and practical applications.

Definition 4 (Projective Lookup Relation). *Let* $(\mathbb{S}, n, \mathbf{t})$ *be a lookup index. We define a projective lookup index as* $\mathrm{I} = \big((\mathbb{S}, n, \mathbf{t}), m, \mathbf{i} = \{i_1, i_2, \ldots, i_n\}\big)$*, and the corresponding projective lookup relation is defined as follows:*

$$\mathrm{PLK}_{\mathrm{I}} = \{\mathbf{w} \in \mathbb{S}^m \,:\, \mathbf{w}_{\mathbf{i}} \subseteq \mathbf{t}\}$$

$\mathbf{w}_{\mathbf{i}}$ *denotes the subset of* $\mathbf{w}$ *indexed by* $\mathbf{i}$*. The projective index* I *is valid if* $(\mathbb{S}, n, \mathbf{t})$ *is valid lookup index and* $\mathbf{i} = \{i_1, i_2, \ldots, i_n\}$ *satisfies* $0 \le i_1 < \cdots < i_n < m$*.*

Similar to the standard lookup relations, we define *projective versions* of the committed lookup relation and the oracle lookup relation in Definitions 5 and 6, respectively.

Definition 5 (Projective Committed Lookup Relation). *Given a projective lookup index* $\mathrm{I} = ((\mathbb{S}, n, \mathbf{t}), m, \mathbf{i} = \{i_1, \ldots, i_n\})$*, the corresponding projective lookup relation* $\mathrm{PLK}_{\mathrm{I}}$*, and a commitment scheme* $(\mathrm{Com}, \mathrm{Open})$ *on* $\mathbb{S}^m$*, the* projective committed lookup relation *is defined as:*

$$\{(c;\, \mathbf{w}) \,:\, \mathbf{w} \in \mathrm{PLK}_{\mathrm{I}},\ c = \mathrm{Com}(\mathbf{w})\}$$

Definition 6 (Projective Oracle Lookup Relation). *Given a projective lookup index* $\mathrm{I} = ((\mathbb{S}, n, \mathbf{t}), m, \mathbf{i} = \{i_1, \ldots, i_n\})$ *and its corresponding relation* $\mathrm{PLK}_{\mathrm{I}}$*, the* projective oracle lookup relation *is defined as:*

$$\{([[\mathbf{w}]];\, \mathbf{w}) \,:\, \mathbf{w} \in \mathrm{PLK}_{\mathrm{I}}\}$$

In the way projective relations are defined above, the indices $i_1, i_2, \ldots, i_n$ are fixed within the witness by embedding them directly into the index. This reflects a common practice in SNARKs, where the statement is typically preprocessed, and we know in advance which witness wires need to be checked for inclusion in a lookup table. Essentially, to prove such a statement, we must compose a projective oracle lookup PIOP with the underlying proof system's PIOP. The central question then arises: since most prior work only addresses the non-projective setting, is it possible to efficiently construct a PIOP for the projective case given an existing PIOP designed for the non-projective setting?

How to Design a PIOP for Projective Lookup Relations? As discussed earlier, lookup tables serve as a useful subprotocol component. The key challenge lies in partially selecting the witness and proving that it appears in a predefined table. A natural approach is to commit to a subvector and demonstrate that it matches the entries at specific indices. However, doing so efficiently is nontrivial, e.g. without constructing a commitment within the arithmetic circuit, concretely, that is because the *shape of the commitment* is changing. In what follows, we explore several potential strategies for how existing techniques—such as the Logup lemma and approaches based on matrix-vector products—can be adapted or extended to support projective lookup efficiently without requiring the explicit construction of the subvector. We extend the Logup lemma to the projective setting (see Lemma 4) and, based on this, generalize `cq` and Protostar to support projective lookups. This approach can also be applied to other lookup arguments that rely on Logup. In Appendix G.3, we explain how to modify Lasso to support projective lookups, with almost no overhead. Another approach, briefly noted in Halo2 [Zca25], is the *polynomial selector* technique. At a high level, the idea is that the prover modifies the witness polynomial so that the indices corresponding to the intended lookups remain unchanged, while all other positions are filled with a trivial element from the lookup table. As a result, verifying membership of this modified witness is equivalent to performing a projective lookup on the original witness.

3.4 Indexed lookup

Indexed lookup [STW24] extends the standard definition of lookup tables. In the traditional lookup setting, there is no requirement for the table $\mathbf{t}$ or the witness $\mathbf{w}$ to be ordered lists—these can even be modeled as a set and a multiset, respectively.[8] The indexed lookup setting introduces two key differences: (I) both the table and the witness are assumed to be ordered lists, also known as vectors, and (II) the verifier additionally receives a commitment to the indices of the witness elements in the table (see Definition 7).

Definition 7 (Indexed Lookup Relation). *Given a* lookup index $\mathrm{I} := (\mathbb{S}, n, \mathbf{t})$*, an* indexed lookup relation *is the set of tuples* $(\mathbf{w}, \mathbf{i})$ *such that* $\mathbf{w} \subseteq \mathbb{S}$ *and* $\mathbf{i} \subseteq [N]$ *where* $|\mathbf{i}| = |\mathbf{w}| = n$*, and for all* $k \in [n]$*, the entry* w_k *equals the table entry at index* i_k*, i.e.,* $w_k = t_{i_k}$*. Formally:*

$$\left\{(\mathbf{w}, \mathbf{i}) \ : \ \forall\, j \in [n],\ w_j = t_{i_j}\right\}.$$

Indexed lookups offer greater expressiveness than standard lookups and enable a wider range of applications. While standard lookups are sufficient for simple tasks such as range checks—ensuring that certain witness values lie within a predefined set—they fall short when implementing arbitrary functions. Consider a function $f : [N] \to \mathbb{S}$, and let $\mathbf{t}_f$ be a table such that the entry at index i is $f(i)$. With indexed lookup tables, the function f can be externally represented via table $\mathbf{t}_f$, avoiding the need to encode f directly in the circuit. This is the core idea behind Jolt [AST24], which constructs tables for all CPU operations (e.g., AND, OR, etc.), resulting in significantly more efficient circuits for general-purpose programs. Similar to the standard lookup relation, we can define the *committed index lookup relation* and the *oracle index lookup relation*, and extend these notions to the projective setting. However, we omit these extensions for simplicity, as they are straightforward.

Generic compiler from standard to indexed Lookup [STW24]. Lasso introduces a general compiler to transform a standard lookup argument into an indexed one when the underlying set is a field with a large enough characteristic, i.e. $\mathbb{S} = \mathbb{F}$. Let $\mathbf{t} \in \mathbb{F}^N$ be a lookup table, and suppose that all table entries lie in the range $[r]$. Assume that $m \cdot N$ is less than the characteristic of the field $\mathbb{F}$. We define a modified table $\mathbf{t}^* \in \mathbb{F}^N$ where each entry is encoded as $t_i^* = i \cdot r + t_i$. For each standard lookup pair (b_j, a_j), define the combined field element $a_j^* = b_j \cdot r + a_j$. A range check must be applied to ensure that $a_j \in [r]$ for each lookup. Under this constraint, the equality $t_{b_j} = a_j$ holds if and only if $t_{b_j}^* = a_j^*$. Conversely, if $t_{b_j} \neq a_j$, then $a_j^* \notin \mathbf{t}^*$. This compiler introduces an additional range check, which increases its cost. As we will see in the next section, some constructions, such as Lasso—natively support indexed lookups, while others, such as constructions based on the Logup lemma, can be extended to handle indexed lookups more efficiently through vector lookups.

Indexed lookup via vector lookups. A natural method to enable indexed lookups is through the use of vector lookups. Suppose we have a table $\mathbf{t}$ indexed by i with entries t_i. If the underlying lookup relation supports vector lookups, we can treat the table as a vector (or tuple) of pairs (i, t_i). This approach can be particularly efficient—for instance, when using the Logup lemma—since it natively supports vector lookups (see Lemma 3) and avoids the need for additional checks, such as range constraints.

3.5 Vector lookup

Vector lookup is a natural generalisation of standard lookup, where each entry in the table is a tuple of elements from $\mathbb{S}$, rather than a single element from $\mathbb{S}$. This concept has appeared in different works [GW20; Hab22; Cam+24; Cho+24; Di+24][9]. We formally define the vector lookup relation in Definition 8.

[8] As discussed in Section 3.1, our definitions treat both the witness $\mathbf{w}$ and table $\mathbf{t}$ as ordered lists because that is the case of interest in proof systems.

[9] These works use different terminology of segment lookup, matrix lookup and vector lookup. We prefer to go with the more natural choice of vector lookup here.

Definition 8 (Vector Lookup Relation). *Given a finite set* $\mathbb{S}$*, a* vector lookup index *is defined as* $\mathrm{I} := (\mathbb{S}, n, k, \mathbf{t})$*, where* n *is the number of lookups, and* $\mathbf{t} \in \left(\mathbb{S}^{(k)}\right)^N$ *is a table consisting of* N *tuples of size* k *over* $\mathbb{S}$*. The corresponding* vector lookup relation *is the set of* k*-tuples over* $\mathbb{S}$ *of size* n *such that each* k*-tuple appears in the table* $\mathbf{t}$*. Formally,*

$$\left\{\mathbf{w} \in \left(\mathbb{S}^{(k)}\right)^n \,\middle|\, \forall i \in [n],\, \mathbf{w}_i \in \mathbf{t}\right\}.$$

Certain techniques, such as the Logup lemma, naturally extend to vectors (see Lemma 3). For instance, MuxProofs [Di+24] and Subplonk [Cho+24] construct their vector lookup arguments based on this extension. Another line of work, introduced in Plonkup and later refined by Campanelli et al. [Cam+24], leverages *homomorphic tables* [EFG22] to realize vector lookups when the underlying lookup table supports homomorphic proofs.

Vector lookup from homomorphic proofs. A straightforward way to support vector lookups over tuples of size k using a lookup argument supporting *aggregatable* proofs (e.g. homomorphic proofs) is to rely on k separate lookup tables and aggregate the lookup proofs. However, even with proof aggregation, the verifier's running time remains linear in k using existing homomorphic table schemes, since the verifier must compute a random linear combination of the proofs. This overhead becomes inefficient for large tuple sizes. An alternative solution for vector lookups, in which the verifier's time does not depend on the tuple size, was suggested by Campanelli et al. [Cam+24]. At a high level, the technique linearizes the lookup table (and subtable) of k-tuples into a table of 3-tuples, where each entry contains the element, its position within the tuple, and its index in the table. For example, given a tuple $\mathbf{r}$ of length k, the transformation produces $\{(x_i, y_j, r_i)\}_{i\in[k],\, j\in[N]}$, where x_i denotes the position of element r_i within the tuple, and y_j denotes the index of the tuple in the lookup table. This transformation requires additional consistency checks to ensure well-formedness, namely that the resulting vector indeed corresponds to a concatenation of k-tuples. For this transformation to be valid, all x_i values must be equal across the tuple. After this transformation and the corresponding correctness checks, the lookup reduces to a lookup over 3-tuples. This can be interpreted as three separate lookup tables, with the proofs then aggregated. Concretely, Campanelli et al. [Cam+24] employs cq, which supports homomorphic proof aggregation.

Generalized vector lookups. In Definition 8, the table can be seen as composed of *tuples*, and our witness corresponds to a vector of tuples in the table. That is, what we are looking up are tuples *in their entirety*. In some settings, we may want to lookup just part of the tuple (and we may not be interested in paying the additional overhead of simply applying a vector lookup); for example, the witness could consist of a pairs $(a_i, b_i)_i$, such that there exists some tuple in the table such that the first element is equal to a_i and the tenth element is equal to b_i. We do not further discuss this more general notion here, but we refer the reader to [Cam+24, Appendix D.1] where the authors define and construct this different notion.

3.6 Online lookup table

In the standard use of lookup tables, the table is fixed and can often be preprocessed. However, in several cases, the table is not fixed during setup but instead depends on the witness. A well-known example is Spark [Set20], which constructs an online table[10] for $\widetilde{\mathsf{eq}}(\mathbf{x}, \mathbf{r})$ over a set of $\mathbf{x} \in \{0,1\}^n$, where $\mathbf{r}$ is the verifier's challenge. Similar ideas arise in distributed proving systems such as Hekaton [Ros+24] and Soloist [Li+25], which generate online lookup tables for mutual witness wires shared across subcircuits. Since these tables depend on the witness, they cannot be preprocessed during the offline phase, making them incompatible with preprocessing schemes such as cq [EFG22]. Exploring additional applications of online lookup tables remains an interesting direction for future work. To formalize this setting, we introduce the following definition of a lookup relation where the table itself appears within the lookup instance rather than in the index. We emphasize that all previous definitions—such as the committed lookup

[10] In Spartan, this is referred to as a *write-once-read-many* memory scheme.

relation, oracle lookup relation, and projective lookup etc.—can be extended to the *online* setting as well. For simplicity, we omit these generalizations here.

Definition 9 (Online Lookup Relation). *Given a finite set $\mathbb{S}$, a lookup index is defined as $\mathrm{I} := (\mathbb{S}, n, N)$, where n is the number of lookups and N is the size of the table. The relation consists of all tuples $(\mathbf{w}, \mathbf{t})$ such that $\mathbf{w} \in \mathbb{S}^n$ and $\mathbf{t} \in \mathbb{S}^N$, with the condition that $w_i \in \mathbf{t}$ for all $i \in [n]$. Equivalently, $\{(\mathbf{w}, \mathbf{t}) \ : \ \mathbf{w} \subseteq \mathbf{t}\}$.*

4 An Overview of Existing Techniques

In this section, we survey existing schemes categorised based on their underlying technique. In Table 1, we give a non-exhaustive list of schemes ordered by techniques together with their compatibility and some asymptotic costs. Given the variety of possible instantiations and thus the complexity of comparing these schemes, our main goal is to give some insight to those who have clear constraints in mind rather than exhibit an overall winner.

Multiset equality technique. Early approaches to (unindexed) lookup arguments namely Arya [Boo+18], Plookup [GW20] and Halo2 [Zca25] are all based on the notion of *multiset equality*. The central idea is that since verifying subset inclusion is more involved than checking multiset equality, these works reduce the problem to a multiset equality check, i.e. $\mathbf{w} \subseteq \mathbf{t}$ if and only if $\mathbf{w} \cup \mathbf{t}$ contains the same elements as $\mathbf{t}$, up to multiplicities. A shared characteristic among these approaches is that the prover's cost depends on the size of the table, making them most efficient when $|\mathbf{t}| \leq |\mathbf{w}|$. For example, bit operations—such as those suggested in *Arya*—are typically performed on small chunks of size 8 or 16 bits. In contrast, systems that decouple the prover's cost from the table size can more efficiently handle bit operations on larger chunks, such as 32 bits. We observe that the underlying multiset equality techniques used in Plookup and Halo2 do not depend on a large field size, and they require no preprocessing (aside from committing to the table and the witness), which makes them suitable for online lookup tables. For example, VerITS [DCB25] applies Plookup over a prime field of size $2^{31} - 1$. RISC0 [ris25b] utilizes Plookup as its underlying lookup argument, demonstrating efficiency even with hash-based polynomial commitment schemes.[11] We review the existing schemes based on the multiset equality notion in Appendix D.

Logup-based. This is a different family of lookup table arguments that reduces a lookup relation into a summation over rational functions. This line of work began with [Eag+24] and was later formalized as a lookup argument in [Hab22]. The core idea is similar to the one used in Arya: To prove that a witness vector $\mathbf{w}$, encoded as polynomial $W(x) = \prod_{i\in[n]}(x + w_i)$, is a subset of a lookup table $\mathbf{t}$, encoded as $T(x) = \prod_{i\in[N]}(x + t_i)$, it suffices to show that there exist non-negative integers $\{m_i\}$ such that $T(x) = \prod_{i\in[N]}(x + t_i)^{m_i}$. However, directly proving this identity leads to high costs, similar to those in Arya. The key insight is that this polynomial identity holds if and only if their logarithmic derivatives match. In other words, for two polynomials g_1 and g_2, we have:

$$g_1(x) = g_2(x) \iff \frac{d}{dx}\log g_1(x) = \frac{d}{dx}\log g_2(x).$$

Substituting $g_1(x)$ and $g_2(x)$ with $W(x)$ and $\prod_{i\in[N]}(x + t_i)^{m_i}$, respectively, we derive the following equivalence:

$$\xrightarrow[g_2(x)=\prod_{i\in[N]}(x+t_i)^{m_i}]{g_1(x)=W(x)} W(x) = \prod_{i\in[N]}(x + t_i)^{m_i} \iff \sum_{i\in[n]}\frac{1}{x + w_i} = \sum_{i\in[N]}\frac{m_i}{x + t_i}.$$

The original work introduces what is referred to as the Logup (Lemma 2), which reformulates the lookup inclusion condition as a rational function identity. One key idea is that verifying

[11] They plan to transition from Plookup to Logup-based techniques in their upcoming version [ris25a].

grand summations is generally simpler than verifying grand products, for example, by using the univariate or standard sumcheck. This is why the Logup formulation of lookups is more widely adopted compared to the grand product approach introduced in Arya. The Logup lemma does not rely on a large field size. For instance, Stwo [Sta25] and SP1 [Lab25], the state-of-the-art STARK-based zkVMs employ Logup for lookups, demonstrating the efficiency of the technique even when instantiated with non-homomorphic PCSs such as FRI-based ones. Existing work such as Logup [Hab22], `cq` [EFG22], and their extensions [PH23; Cam+24; ZSG24; Dor24; BC24; CFG25] build on the Logup lemma; for a detailed overview, see Appendix F.

Subvector extraction (matrix-vector) technique. One general technique to verify a lookup relation is *subvector extraction*, commonly implemented as a matrix–vector multiplication, i.e. a sparse constraint system of rank one. The high-level idea is as follows: if the witness $\mathbf{w}$ is included in the table $\mathbf{t}$, then for every w_i that equals t_j, there exists an elementary vector e_j of the same length as $\mathbf{t}$ such that $e_j \cdot \mathbf{t} = w_i$. Here, e_j is zero everywhere except at index j, where it is equal to 1. By concatenating these elementary vectors as rows, we obtain an *elementary matrix* $M_{n\times N}$, where each row has exactly one nonzero entry equal to 1. The lookup relation is then equivalent to the matrix equation $M \times \mathbf{t} = \mathbf{w}$. With this new interpretation, the verifier is given commitments to a table $\mathbf{t}$ and a witness $\mathbf{w}$. The prover's goal is to prove that there exists an elementary matrix M such that $M \times \mathbf{t} = \mathbf{w}$. The indexed lookup naturally generalizes to the case where the verifier is also given a commitment to the index vector $(i_1, i_2, \ldots, i_n)$, which satisfies $t_{i_1} = w_1$, $t_{i_2} = w_2$, ... The corresponding *map* matrix M for this index vector is

$$M = \begin{bmatrix} e_{i_1} \\ e_{i_2} \\ \vdots \\ e_{i_n} \end{bmatrix},$$

where e_{i_k} denotes the standard basis vector with a 1 in position i_k and zeros elsewhere. Finally, the prover proves that this matrix M, derived from the committed index vector, indeed satisfies $M \times \mathbf{t} = \mathbf{w}$. Existing approaches based on this technique include Caulk [Zap+22a] and Caulk+ [PK22], which perform subvector extraction implicitly, as well as Baloo [Zap+22b], Lasso [STW24], and Shout [ST25], which carry out the extraction explicitly using the matrix–vector technique. The essence of the matrix–vector technique does not rely on a large field size; for instance, Lasso has been adapted to binary fields [DP25] as well as to ring elements [Cry]. For an overview of these approaches, see Appendix G.

Accumulator-based. Another approach to building lookup arguments is through set accumulators that support *commit-and-prove batch openings*. In a nutshell, these accumulators allow one to succinctly commit to the table $\mathbf{t}$ and to a subtable $\mathbf{t}'$, and to generate a succinct proof that $\mathbf{t}' \subseteq \mathbf{t}$. Flookup [GK22] follows this approach, with pairing-based accumulators, to extract the subtable of all witness entries and then gives a protocol to show that these entries of the committed witness are in the subtable. Other works [Cam+22a; Han+25] instead rely on RSA accumulators, in which the table must be encoded with prime numbers, and then use specialized protocols to efficiently link RSA accumulators with SNARKs.

Index preprocessing. One technique of theoretical interest used to build lookup arguments is polynomial preprocessing [Cam+25]. To provide background, in Lasso [STW24], the authors introduce a variant of their protocol called *generalized Lasso* that constructs a lookup PIOP using a sparse sumcheck protocol. However, this construction relies critically on the fact that the *table is structured* and can therefore its multilinear extension can be evaluated efficiently—meaning it only applies to a restricted class of tables. The key idea in Campanelli et al. [Cam+25] is to preprocess *any* table so that its evaluations can be performed efficiently, in sublinear time with respect to the size of the polynomial (or table). This allows them to employ generalized Lasso to design an efficient lookup PIOP even for unstructured tables. The polynomial preprocessing technique builds on prior works [KU08; KU11], which, at a high level, show that given a polynomial, one can preprocess it so that evaluations can be computed in sublinear time while requiring only polylogarithmic storage. The authors note, however, that

their approach is unlikely to be immediately practical due to the large constants involved in the specific preprocessing scheme [KU11].

Lookup accumulation. Accumulation schemes are an efficient way to build recursive proof systems such as Incremental Verifiable Computation (IVC) [Val08]. The use of lookup tables in the context of recursive SNARKs is non-trivial. For example, using a lookup argument within a single IVC step typically requires one of the following strategies: (I) *Encoding the lookup verification directly into the step circuit.* This can substantially increase the circuit size and cost, especially when the underlying lookup verifier involves expensive operations such as pairings—as is the case with KZG-based lookup arguments like cq. (II) *Accumulating the lookup verifier using a running accumulator.* In this case, the verifier logic for the lookup argument is not encoded in the circuit. Instead, only the verification logic for the accumulation lookup is encoded, which can significantly reduce overhead by, for example, avoiding pairing operations. In Appendix H, we review existing techniques for accumulating lookup relations, namely Protostar [BC23], nLookup [KS24] and FLI [GM24], which enable the efficient integration of specific lookup methods into the *recursive proofs setting*. An overview of these schemes is presented in Table 3.

5 Considerations on Lookup Arguments in Practice

In order to use lookup arguments in practice, a first crucial consideration is *compatibility* between the lookup scheme and the underlying proof system. This compatibility manifests itself in two main ways. First, the lookup scheme and the proof system must operate over the same finite structure, e.g. the same finite field. For example, a proof system such as Plonky2 [Pol22], which operates over a small prime field, is not compatible with KZG-based constructions that require a large prime field. Second, their composition must be structurally compatible: the two systems should either share the same commitment when composed at the argument level, or rely on the same polynomial commitment scheme when the PIOPs are compiled into arguments. For example, if the lookup PIOP uses a univariate polynomial commitment scheme like KZG, but the proof system relies on multivariate commitments, then composing these two may require simultaneously committing to the same witness in both univariate and multivariate forms. Proving that these two commitments correspond to the same underlying witness within an arithmetic circuit can be highly inefficient, and general compilers [Zha+20; Boo+22] would add some extra overhead. To address such challenges, recent works propose scheme adaptations. For example, μ-seek [CFG25] enables cq to operate with witnesses committed via multilinear polynomial commitments (see Appendix F.2), making cq compatible with proof systems based on multilinear polynomials such as HyperPlonk. Another idea to solve this problem might be to link the a univariate polynomial to a multivariate one at the commitment level; such relevant primitive as been called *dual polynomial commitment* [GNS24] and leverage some properties[12] of specific polynomials to map efficiently from a univariately committed witness to the same one committed using a multivariate basis. We leave the study of this primitive in the context of lookup compatibility to future work.

When selecting among compatible schemes, important considerations are the table size, its potential for decomposition, efficiency of the scheme, and whether a lookup table offers greater efficiency than a naive implementation.

Table size. A crucial factor in the efficiency of lookup protocols is table size. Many schemes are designed under the assumption that the table is significantly larger than the number of lookups. In such cases, the solution is to use *table-efficient* lookup arguments in which the running time of the prover grows sublinearly in the table size. Existing table-efficient schemes can be divided into two families (cf. Figure 1). The first family achieves table-efficiency relying on some structure of the large table (e.g., if it can be reconstructed with sublinear computation from smaller tables). In this setting the family of constructions based on Lasso [STW24;

[12] Such as a specific linear isomorphism between a univariate Lagrange basis over a set of roots of unity and a multivariate Lagrange basis over the boolean hypercube.

CFR25; ST25] offers the best tradeoffs between proving time and other performance metrics. The second family achieves table-efficiency for arbitrary, unstructured tables, by means of a preprocessing stage that produces auxiliary material which the prover can use to generate lookup proofs in time sublinear in the table size. We observe a few patterns in the efficiency profile of existing preprocessing-based lookups. One construction [Cam+25] obtains table efficiency via preprocessing algorithms for polynomial evaluation; this construction is generic (e.g., they can be realized only from hash functions) but mostly of theoretical interest since preprocessing and proving time have very large constants. Several schemes—whose state of the art is represented by the cq family [EFG22; Cam+24; CFG25; ZSG24] of constructions—achieve table-efficiency relying on specific properties of pairing-based cryptography, including a preprocessing method for computing in quasilinear time the membership proofs of all the elements of a committed table. These schemes have extremely fast provers but their preprocessing has concrete scalability limitations. For instance, for a table of size 2^{32}, preprocessing requires around 2^{37} group operations—an effort that would take days even on a high-performance server[13]. Finally, a different line of work achieves a better trade-off by committing to the table using a lightweight accumulator (e.g., a Merkle tree), precomputing membership proofs for all elements, and then proving lookups by proving the existence of the relevant subset of membership proofs with a general-purpose SNARK. These constructions are called non-black-box, since the membership check must be encoded in the circuit; an early instance of this method appears in Pantry [Bra+13].

This state of affairs raises several open questions in this line of work, such as whether pairing-based performance can be replicated under alternative assumptions, or if one can design black-box solutions that offer a balanced profile where both proving and preprocessing remain feasible for larger tables.

Applications. Precomputed tables are a well-established technique in computer science to accelerate computationally intensive operations. By storing the results of commonly used or complex computations ahead of time, systems can replace expensive runtime calculations with efficient table lookups. In proof systems, lookup tables serve a similar role: they reduce computation by replacing *repetitive or complex operations* with direct access to precomputed values. Rather than embedding such operations directly into the arithmetic circuit, the prover can read the result from a pre-defined lookup table. This strategy decreases both the size and cost of the circuit, resulting in more efficient proof generation. However, this efficiency comes at the cost of an additional requirement: the prover must now demonstrate that the lookup accesses are valid. This trade-off is typically favorable for two key reasons. First, thanks to the preprocessing explained in the previous paragraph, most of the work is pushed offline. Second, lookup systems support batching, enabling multiple lookup operations to be verified in a single invocation of the argument. This section provides an overview of how lookup tables are used in practice.

We organize these applications into the following main categories: (I) The biggest application of lookup tables in practice is for reducing constraint count for replacing non-native or not arithmetic-friendly operations with lookup operations [Sze+23; Gra+22; AST24; Woo+25; SLZ24; Xin+25; DCB25; Kan+22; Xio+22; Der+23; Lu+24; GFN24; Sou24]. Such non-native or non-arithmetic-friendly operations may include operations such as bit decomposition, range proofs, value comparison, floating-point operations and other functions that use such operations as their primitives, such as hash functions. In this case, the table is often small or structured, and the Lasso family of techniques shines. (II) Another less-explored application of lookup arguments is set membership, where one proves that all elements in a commitment belong to a given set or vector. This technique is advantageous in scenarios where you want to access a large and unstructured set, e.g. a set of public keys. In this case, it is interesting to use a scheme with the table-efficiency property. (III) Lookups are used in the context of memory correctness [Zha+23; Dut+24] and state machines and finite automata [Ang+24; Di+24] too, e.g. to prove that the execution of the machine adheres to its transition rules. In this case, the table is big and unstructured, leaning towards indexed lookup arguments supporting precomputations.

[13] https://zka.lc/

Technique	Scheme/Family	T_{precomp}	$\lvert\pi\rvert$	T_{prove}	T_{vfy}	Compatibility
Multisets	Plookup [GW20]	—	const.	$(N+n)\log(N+n)$	const.	uni/multivar.
	Halo2 [Zca25]	—	const.	$N \log N$	const.	univar.
Logup	Logup [PH23]	—	const.	$N \log N$	const.	multivar.
	Logup+GKR [EFG22]	—	$\log(N+n)$	$N+n$	$\log(N+n)$	univar.
	cq† [EFG22; Cam+24]	$N \log N$	const.	$n \log n$	const.	KZG
Subvector Extractable	Caulk† [Zap+22a]	$N \log N$	const.	$n^2+n \log N$	const.	KZG
	Caulk+† [PK22]	$N \log N$	const.	n^2	const.	KZG
	Baloo† [Zap+22b]	$N \log N$	const.	$n \log^2 n$	const.	KZG
	Lasso† [STW24; CFR25]	—	$\log^2 n$	$N+n, cn^\star$	$\log^2 n$	multivar.
	Shout [ST25]	—	$d \log n$	$cn^\star$	$d \log n$	multivar.
Accumulator	Flookup [GK22]	$N \log N$	const.	$n \log^2 n$	const.	KZG
	Duplex† [Cam+22a; Han+25]	$N \log N$	const.	$n \log n$	const.	Pedersen

N: table size n: number of lookups †: supports zero-knowledge $\star$: structured tables only
On compatibility: Pedersen $\supseteq$ KZG, univariate $\supseteq$ KZG (i.e., KZG is a special case of both)

Table 1: Asymptotic comparison of lookup argument schemes (Big-Oh is implicit). N is the size of the table and n is the number of lookups. For Lasso, we consider costs as it is initiated with Dory [Lee20], the prover time considers two different cases when the table is structured and unstructured.

It is interesting to mention that, as shown by Dutta et al. [Dut+24], when the memory also supports write operations, known approaches fail because they require some updatable table. How to build such a scheme with an efficiency that is close to the state-of-the-art for classical preprocessing is still an open question. (IV) A theoretically interesting application of lookup tables is strengthening an extractor by proving the witness belongs to a finite set. Zinc [Gar+25] employs a novel use of lookup PIOP to strengthen the extractor of its PCS. At a high level, Zinc aims to build an argument of knowledge for an integer witness, while the underlying PCS extractor is only capable of extracting bounded rational numbers. Since knowledge soundness requires the extraction of an integer witness, Zinc composes its PIOP with a lookup PIOP, ensuring that the witness indeed consists of integers. When combined with the PCS extractor, this composition ensures the extraction of a satisfying integer witness.[14]

Projective ratio. One last key consideration in the use of lookup tables is the proportion of the witness that benefits from the lookup argument. For instance, to prove a single range proof with a lookup argument might introduce more overhead than the benefit it provides when using a lookup argument. Verifying that a portion of the witness is contained in a table requires *projective lookups*, which may still incur a cost linear in the total size of the witness—e.g. for witness selection. Therefore, if the portion of the witness subject to lookup is relatively small compared to the total witness size, employing a lookup argument may not yield a meaningful efficiency gain.

6 Lookup Arguments vs Other Primitives

Set accumulator and vector commitments. Set accumulators [BM94; BP97; CL02] and vector commitments [CF13] are cryptographic primitives that allow committing to sets and vectors, respectively—and later proving the inclusion of elements or values. Possibly, they may support batch openings, enabling a single proof to attest to multiple inclusions. These primitives are closely related to the *lookup relations*. The key distinction between a lookup protocol

[14] We stress, however, that the resulting complexity for computations with with a vector of integers of size n, where each element is of size $\approx 2^B$ yields a lookup argument where the verifier runs in time $\Omega(\mathsf{max}(2^{B/2}, \sqrt{n})$. For applications with large integers or those with "long" witnesses (large n), this may not be practical. An alternative is offered by the work in [CHA24], which provides a Spartan-like construction for integer relations with better asymptotics (growing with $o(\log^2(B) \cdot \log^2(n))$) and it should be possible to extend it directly to lookup relations with similar asymptotics following [STW24].

and a set accumulator or vector commitment is the existence of duplicates. A lookup witness may contain duplicate elements, whereas in set membership proofs or subvector opening, the elements are unique, and duplication does not occur. Another important difference is that in lookup relations, the verifier holds a commitment to the witness, rather than receiving it in plaintext. In this sense, lookup relations generalize these primitives by introducing commit-and-prove functionality on both sides of the relation. Moreover, in lookup relations the table is commonly[15] assumed to be public, whereas in set accumulators and vector commitments, the underlying set or vector may remain private[16].

Memory checking. A related concept to lookup tables is *memory checking* [Blu+91; Set+18; ST25]. At a high level, memory checking allows a prover to demonstrate that a transcript of memory reads and writes is consistent—for example, that every read returns the value from the most recent write. One key difference from lookups is that memory checking supports *write* operations, whereas lookup tables can be viewed as read-only memory. In some works, write operations are leveraged to construct online lookup tables [Xio+22; Ros+24], for instance, when the table depends on the verifier's challenge. Memory checking is a broad research area, and we do not cover it in detail here. For an overview of memory-checking techniques, we refer the reader to Appendix B of Jolt [AST24].

Updateable lookup tables. A notable subclass of lookup tables, closely related to memory-checking, is the class of *updateable lookup tables* [Dut+24; Han+24]. In contrast to conventional preprocessing tables—where any modification invalidates the preprocessing phase—updateable lookup tables are specifically designed to permit modifications after their initial construction. It is important to distinguish updateable lookup tables from online lookup tables: while the latter are constructed once during the online phase and remain fixed thereafter, updateable lookup tables may be precomputed offline, yet can be modified multiple times during the online phase.

[15] See exceptions we discuss in Section 3.6.

[16] The literature on accumulators and vector commitments is vast. Besides the works we cite in other parts of the document, partial lists of relevant works are: for accumulators [BBF19; Lip12; CHAK23; Sri+22; SGS20; CHAK24; PTT11]; for vector commitments [LM19; WW23; Lib24; Gor+20; PPS21; Cam+22b; Cam+20; Bot+25; CEO22]

Acknowledgments

Hossein Hafezi conducted this work while hosted by Dario Fiore at IMDEA Software. We would like to thank, in alphabetical order, Arasu Arun, Joseph Bonneau, Benedikt Bünz, Albert Garreta, Abhiram Kothapalli, Shahar Papini, Mohammadamin Raisi, Alireza Shirzad, and Aranxta Zapico for their comments and opinions.

This work is supported by the PICOCRYPT project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001283), partially supported by projects PRODIGY (TED2021-132464B-I00), ESPADA (PID2022-142290OB-I00) and CEX2024-001471-M funded by MCIN/AEI/10.13039/501100011033/.

References

[a1625] a16z Crypto. *a16z/jolt: The simplest and most extensible zkVM*. https://github.com/a16z/jolt. Accessed: August 15, 2025. 2025.

[a1624] possibly Justin Thaler etal. a16z Crypto ("JoltBook"). *Instruction lookups*. https://jolt.a16zcrypto.com/how/instruction_lookups.html. Part of the JoltBook documentation on Jolt's lookup-based instruction execution (Lasso lookup argument). 2024.

[Ang+24] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. "Reef: fast succinct non-interactive zero-knowledge regex proofs". In: *Proceedings of the 33rd USENIX Conference on Security Symposium*. SEC '24. Philadelphia, PA, USA: USENIX Association, 2024. ISBN: 978-1-939133-44-1.

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. "Jolt: SNARKs for Virtual Machines via Lookups". In: *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part VI*. Zurich, Switzerland: Springer-Verlag, 2024, 3–33. ISBN: 978-3-031-58750-4. DOI: 10.1007/978-3-031-58751-1_1. URL: https://doi.org/10.1007/978-3-031-58751-1_1.

[BP97] Niko Barić and Birgit Pfitzmann. "Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees". In: *Advances in Cryptology — EUROCRYPT '97*. Ed. by Walter Fumy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 480–494. ISBN: 978-3-540-69053-5.

[BS+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. "Aurora: Transparent Succinct Arguments for R1CS". In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Cham: Springer International Publishing, 2019, pp. 103–128. ISBN: 978-3-030-17653-2.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive Oracle Proofs". In: *Proceedings, Part II, of the 14th International Conference on Theory of Cryptography - Volume 9986*. Berlin, Heidelberg: Springer-Verlag, 2016, 31–60. ISBN: 9783662536438. DOI: 10.1007/978-3-662-53644-5_2. URL: https://doi.org/10.1007/978-3-662-53644-5_2.

[BM94] Josh Benaloh and Michael de Mare. "One-Way Accumulators: A Decentralized Alternative to Digital Signatures". In: *Advances in Cryptology — EUROCRYPT '93*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 274–285. ISBN: 978-3-540-48285-7.

[Bit+13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "Recursive composition and bootstrapping for SNARKS and proof-carrying data". In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '13. Palo Alto, California, USA: Association for Computing Machinery, 2013, 111–120. ISBN: 9781450320290. DOI: 10.1145/2488608.2488623. URL: https://doi.org/10.1145/2488608.2488623.

[Blu+91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. "Checking the correctness of memories". In: *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. SFCS '91. San Juan, Puerto Rico: IEEE Computer Society, 1991, 90–99. ISBN: 0818624450. DOI: 10.1109/SFCS.1991.185352. URL: https://doi.org/10.1109/SFCS.1991.185352.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. "Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains". In: 2019, pp. 561–586. DOI: 10.1007/978-3-030-26948-7_20.

[Bon+20a] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. *Efficient polynomial commitment schemes for multiple points and polynomials*. Cryptology ePrint Archive, Paper 2020/081. 2020. URL: https://eprint.iacr.org/2020/081.

[Bon+20b] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. *Mina: Decentralized Cryptocurrency at Scale*. Tech. rep. 2020/352. Cryptology ePrint Archive, 2020. URL: https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf.

[Boo+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. "Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution". In: *Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I*. Brisbane, QLD, Australia: Springer-Verlag, 2018, 595–626. ISBN: 978-3-030-03325-5. DOI: 10.1007/978-3-030-03326-2_20. URL: https://doi.org/10.1007/978-3-030-03326-2_20.

[Boo+22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrú. "Gemini: Elastic SNARKs for Diverse Environments". In: *Advances in Cryptology – EUROCRYPT 2022*. Ed. by Orr Dunkelman and Stefan Dziembowski. Cham: Springer International Publishing, 2022, pp. 427–457. ISBN: 978-3-031-07085-3.

[Bot+25] Vincenzo Botta, Simone Bottoni, Matteo Campanelli, Emanuele Ragnoli, and Alberto Trombetta. *qedb: Expressive and Modular Verifiable Databases (without SNARKs)*. Cryptology ePrint Archive, Paper 2025/1408. 2025. URL: https://eprint.iacr.org/2025/1408.

[Bra+13] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. "Verifying computations with state". In: *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 341–357.

[BH11] Johannes Buchmann and Safuat Hamdy. "A survey on IQ cryptography". In: *Public-Key Cryptography and Computational Number Theory*. 2011, pp. 1–15.

[BC23] Benedikt Bünz and Binyi Chen. "Protostar: Generic Efficient Accumulation/Folding for Special-Sound Protocols". In: 2023, pp. 77–110. DOI: 10.1007/978-981-99-8724-5_3.

[Bün+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. "Proof-Carrying Data Without Succinct Arguments". In: 2021, pp. 681–710. DOI: 10.1007/978-3-030-84242-0_24.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. "Transparent SNARKs from DARK Compilers". In: 2020, pp. 677–706. DOI: 10.1007/978-3-030-45721-1_24.

[BC24] Benedikt Bünz and Jessica Chen. "Proofs for Deep Thought: Accumulation for Large Memories and Deterministic Computations". In: Dec. 2024, pp. 269–301. ISBN: 978-981-96-0934-5. DOI: 10.1007/978-981-96-0935-2_9.

[CL02] Jan Camenisch and Anna Lysyanskaya. "Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials". In: *Advances in Cryptology — CRYPTO 2002*. Ed. by Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 61–76. ISBN: 978-3-540-45708-4.

[Cam+25] Matteo Campanelli, Mario Carrillo, Ignacio Cascudo, Dario Fiore, Danilo Francati, and Rosario Gennaro. *On the Power of Polynomial Preprocessing: Proving Computations in Sublinear Time, and More*. Cryptology ePrint Archive, Paper 2025/238. 2025. URL: https://eprint.iacr.org/2025/238.

[CEO22] Matteo Campanelli, Felix Engelmann, and Claudio Orlandi. "Zero-Knowledge for Homomorphic Key-Value Commitments with Applications to Privacy-Preserving Ledgers". In: 2022, pp. 761–784. DOI: 10.1007/978-3-031-14791-3_33.

[Cam+24] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. "Lookup Arguments: Improvements, Extensions and Applications to Zero-Knowledge Decision Trees". In: 2024, pp. 337–369. DOI: 10.1007/978-3-031-57722-2_11.

[CFR25] Matteo Campanelli, Antonio Faonio, and Luigi Russo. "SNARKs for Virtual Machines Are Non-malleable". In: *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV*. Ed. by Serge Fehr and Pierre-Alain Fouque. Vol. 15604. Lecture Notes in Computer Science. Springer, 2025, pp. 153–183. DOI: 10.1007/978-3-031-91134-7_6. URL: https://doi.org/10.1007/978-3-031-91134-7_6.

[CFG25] Matteo Campanelli, Dario Fiore, and Rosario Gennaro. "Natively Compatible Super-Efficient Lookup Arguments and How to Apply Them: Natively Compatible Super-Efficient Lookup Arguments". In: *J. Cryptol.* 38.1 (Jan. 2025). ISSN: 0933-2790. DOI: 10.1007/s00145-024-09535-0. URL: https://doi.org/10.1007/s00145-024-09535-0.

[Cam+20] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. "Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage". In: 2020, pp. 3–35. DOI: 10.1007/978-3-030-64834-3_1.

[Cam+22a] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. "Succinct Zero-Knowledge Batch Proofs for Set Accumulators". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 455–469. ISBN: 9781450394505. DOI: 10.1145/3548606.3560677. URL: https://doi.org/10.1145/3548606.3560677.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: 2019, pp. 2075–2092. DOI: 10.1145/3319535.3339820.

[CHA24] Matteo Campanelli and Mathias Hall-Andersen. *Fully Succinct Arguments over the Integers from First Principles*. Cryptology ePrint Archive, Paper 2024/1548. 2024. URL: https://eprint.iacr.org/2024/1548.

[CHAK24] Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. "Curve forests: Transparent zero-knowledge set membership with batching and strong security". In: *Cryptology ePrint Archive* (2024).

[CHAK23] Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. "Curve Trees: Practical and Transparent Zero-Knowledge Accumulators". In: 2023, pp. 4391–4408.

[Cam+22b] Matteo Campanelli, Anca Nitulescu, Carla Ràfols, Alexandros Zacharakis, and Arantxa Zapico. "Linear-Map Vector Commitments and Their Practical Applications". In: 2022, pp. 189–219. DOI: 10.1007/978-3-031-22972-5_7.

[CF13] Dario Catalano and Dario Fiore. "Vector Commitments and Their Applications". In: *Public-Key Cryptography – PKC 2013*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 55–72. ISBN: 978-3-642-36362-7.

[Che+23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: 2023, pp. 499–530. DOI: 10.1007/978-3-031-30617-4_17.

[Che+20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. *Reducing Participation Costs via Incremental Verification for Ledger Systems*. Cryptology ePrint Archive, Paper 2020/1522. 2020. URL: https://eprint.iacr.org/2020/1522.

[Chi+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: 2020, pp. 738–768. DOI: 10.1007/978-3-030-45721-1_26.

[CT10] Alessandro Chiesa and Eran Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards". In: 2010, pp. 310–331.

[Cho+24] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. "SublonK: Sublinear Prover PlonK." In: *Proc. Priv. Enhancing Technol.* 2024.3 (2024), pp. 314–335. URL: http://dblp.uni-trier.de/db/journals/popets/popets2024.html#ChoudhuriGGSS24.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. "Practical verified computation with streaming interactive proofs". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, 90–112. ISBN: 9781450311151. DOI: 10.1145/2090236.2090245. URL: https://doi.org/10.1145/2090236.2090245.

[DCB25] Trisha Datta, Binyi Chen, and Dan Boneh. "VerITAS: Verifying Image Transformations at Scale". In: May 2025, pp. 4606–4623. DOI: 10.1109/SP61157.2025.00097.

[DL78] Richard A. Demillo and Richard J. Lipton. "A probabilistic remark on algebraic program testing". In: *Information Processing Letters* 7.4 (1978), pp. 193–195. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(78)90067-4. URL: https://www.sciencedirect.com/science/article/pii/0020019078900674.

[Der+23] Tal Derei et al. "Scaling Zero-Knowledge to Verifiable Databases". In: *Proceedings of the 1st Workshop on Verifiable Database Systems*. VDBS '23. Seattle, WA, USA: Association for Computing Machinery, 2023, 1–9. ISBN: 9798400707759. DOI: 10.1145/3595647.3595648. URL: https://doi.org/10.1145/3595647.3595648.

[Di+24] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. "MuxProofs: Succinct Arguments for Machine Computation from Vector Lookups". In: *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings. Part V*. Kolkata, India: Springer-Verlag, 2024, 236–265. ISBN: 978-981-96-0934-5. DOI: 10.1007/978-981-96-0935-2_8. URL: https://doi.org/10.1007/978-981-96-0935-2_8.

[DP25] Benjamin E. Diamond and Jim Posen. "Succinct Arguments over Towers of Binary Fields". In: *Advances in Cryptology – EUROCRYPT 2025*. Ed. by Serge Fehr and Pierre-Alain Fouque. Cham: Springer Nature Switzerland, 2025, pp. 93–122. ISBN: 978-3-031-91134-7.

[Dor24] Tesseract Dore. *TaSSLE: Lasso for the commitment-phobic*. Cryptology ePrint Archive, Paper 2024/1075. 2024. URL: https://eprint.iacr.org/2024/1075.

[Dut+24] Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. "Batching-Efficient RAM using Updatable Lookup Arguments". In: Dec. 2024, pp. 4077–4091. DOI: 10.1145/3658644.3670356.

[EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. *cq: Cached quotients for fast lookups*. Cryptology ePrint Archive, Paper 2022/1763. 2022. URL: https://eprint.iacr.org/2022/1763.

[Eag+24] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. "Bulletproofs++: Next Generation Confidential Transactions via Reciprocal Set Membership Arguments". In: *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*. Zurich, Switzerland: Springer-Verlag, 2024, 249–279. ISBN: 978-3-031-58739-9. DOI: 10.1007/978-3-031-58740-5_9. URL: https://doi.org/10.1007/978-3-031-58740-5_9.

[FK23] Dankrad Feist and Dmitry Khovratovich. *Fast amortized KZG proofs*. Cryptology ePrint Archive, Paper 2023/033. 2023. URL: https://eprint.iacr.org/2023/033.

[GK22] Ariel Gabizon and Dmitry Khovratovich. *flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size*. Cryptology ePrint Archive, Paper 2022/1447. 2022. URL: https://eprint.iacr.org/2022/1447.

[GW20] Ariel Gabizon and Zachary J. Williamson. *plookup: A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Paper 2020/315. 2020. URL: https://eprint.iacr.org/2020/315.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[GNS24] Chaya Ganesh, Vineet Nair, and Ashish Sharma. "Dual Polynomial Commitment Schemes and Applications to Commit-and-Prove SNARKs". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, 884–898. ISBN: 9798400706363. DOI: 10.1145/3658644.3690219. URL: https://doi.org/10.1145/3658644.3690219.

[GM24] Albert Garreta and Ignacio Manzur. "FLI: Folding Lookup Instances". In: *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings. Part V*. Kolkata, India: Springer-Verlag, 2024, 402–435. ISBN: 978-981-96-0934-5. DOI: 10.1007/978-981-96-0935-2_13. URL: https://doi.org/10.1007/978-981-96-0935-2_13.

[Gar+25] Albert Garreta, Hendrik Waldner, Katerina Hristova, and Luca Dall'Ava. Zinc*: Succinct Arguments with Small Arithmetization Overheads from IOPs of Proximity to the Integers*. Cryptology ePrint Archive, Paper 2025/316. 2025. URL: https://eprint.iacr.org/2025/316.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating Computation: Interactive Proofs for Muggles". In: *J. ACM* 62.4 (Sept. 2015). ISSN: 0004-5411. DOI: 10.1145/2699436. URL: https://doi.org/10.1145/2699436.

[Gor+20] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. "Pointproofs: Aggregating Proofs for Multiple Vector Commitments". In: 2020, pp. 2007–2023. DOI: 10.1145/3372297.3417244.

[Gra+22] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. "Reinforced Concrete: A Fast Hash Function for Verifiable Computation". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 1323–1335. ISBN: 9781450394505. DOI: 10.1145/3548606.3560686. URL: https://doi.org/10.1145/3548606.3560686.

[Gro16] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11.

[GFN24] Binbin Gu, Juncheng Fang, and Faisal Nawab. *PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification*. 2024. arXiv: 2411.15031 [cs.DB]. URL: https://arxiv.org/abs/2411.15031.

[Hab22] Ulrich Haböck. *Multivariate lookups based on logarithmic derivatives*. Cryptology ePrint Archive, Paper 2022/1530. 2022. URL: https://eprint.iacr.org/2022/1530.

[HLP24] Ulrich Haböck, David Levit, and Shahar Papini. *Circle STARKs*. Cryptology ePrint Archive, Paper 2024/278. 2024. URL: https://eprint.iacr.org/2024/278.

[Haf+25] Hossein Hafezi, Alireza Shirzad, Benedikt Bünz, and Joseph Bonneau. *IronDict: Transparent Dictionaries from Polynomial Commitments*. Cryptology ePrint Archive, Paper 2025/1580. 2025. URL: https://eprint.iacr.org/2025/1580.

[Han+24] Semin Han, Geonho Yoon, Hyunok Oh, and Jihye Kim. *DUPLEX: Scalable Zero-Knowledge Lookup Arguments over RSA Group*. Cryptology ePrint Archive, Paper 2024/1509. 2024. URL: https://eprint.iacr.org/2024/1509.

[Han+25] Semin Han, Geonho Yoon, Hyunok Oh, and Jihye Kim. "DUPLEX: Scalable Zero-Knowledge Lookup Arguments over RSA Group". In: *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, 72–86. ISBN: 9798400714108. DOI: 10.1145/3708821.3733863. URL: https://doi.org/10.1145/3708821.3733863.

[Kad+25] George Kadianakis, Arantxa Zapico, Hossein Hafezi, and Benedikt Bünz. *KZH-Fold: Accountable Voting from Sublinear Accumulation*. Cryptology ePrint Archive, Paper 2025/144. 2025. URL: https://eprint.iacr.org/2025/144.

[Kan+22] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. *Scaling up Trustless DNN Inference with Zero-Knowledge Proofs*. 2022. arXiv: 2210.08674 [cs.CR]. URL: https://arxiv.org/abs/2210.08674.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8_11.

[KU08] Kiran S. Kedlaya and Christopher Umans. "Fast Modular Composition in any Characteristic". In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 2008, pp. 146–155. DOI: 10.1109/FOCS.2008.13.

[KU11] Kiran S. Kedlaya and Christopher Umans. "Fast Polynomial Factorization and Modular Composition". In: *SIAM Journal on Computing* 40.6 (2011), pp. 1767–1802. DOI: 10.1137/08073408X. eprint: https://doi.org/10.1137/08073408X. URL: https://doi.org/10.1137/08073408X.

[Kos+20] Ahmed Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. "MIRAGE: succinct arguments for randomized algorithms with applications to universal zk-SNARKs". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *Advances in Cryptology – CRYPTO 2022*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Cham: Springer Nature Switzerland, 2022, pp. 359–388. ISBN: 978-3-031-15985-5.

[KS24] Abhiram Kothapalli and Srinath T. V. Setty. "HyperNova: Recursive Arguments for Customizable Constraint Systems". In: 2024, pp. 345–379. DOI: 10.1007/978-3-031-68403-6_11.

[Lab25] Succinct Labs. *SP1: a zero-knowledge virtual machine for RISC-V*. https://github.com/succinctlabs/sp1. Accessed: 2025-10-05. 2025.

[LM19] Russell W. F. Lai and Giulio Malavolta. "Subvector Commitments with Application to Succinct Arguments". In: 2019, pp. 530–560. DOI: 10.1007/978-3-030-26948-7_19.

[Lee20] Jonathan Lee. "Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1274. URL: https://api.semanticscholar.org/CorpusID:222939305.

[Li+25] Weihan Li, Zongyang Zhang, Yun Li, Pengfei Zhu, Cheng Hong, and Jianwei Liu. *Soloist: Distributed SNARKs for Rank-One Constraint System*. Cryptology ePrint Archive, Paper 2025/557. 2025. URL: https://eprint.iacr.org/2025/557.

[Lib24] Benoît Libert. "Vector Commitments with Proofs of Smallness: Short Range Proofs and More". In: 2024, pp. 36–67. DOI: 10.1007/978-3-031-57722-2_2.

[Lip12] Helger Lipmaa. "Secure Accumulators from Euclidean Rings without Trusted Setup". In: 2012, pp. 224–240. DOI: 10.1007/978-3-642-31284-7_14.

[Lu+24] Tao Lu et al. *An Efficient and Extensible Zero-knowledge Proof Framework for Neural Networks*. Cryptology ePrint Archive, Paper 2024/703. 2024. URL: https://eprint.iacr.org/2024/703.

[Lun+92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. "Algebraic methods for interactive proof systems". In: *J. ACM* 39.4 (Oct. 1992), 859–868. ISSN: 0004-5411. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[Nex24] Nexus Collective. *Nexus zkVM 1.0 Whitepaper: A Planetary-Scale Verifiable Compute Network*. Whitepaper. Accessed: 2025-07-06. Online: Nexus, Jan. 2024.

[Ngu05] Lan Nguyen. "Accumulators from Bilinear Pairings and Applications". In: 2005, pp. 275–292. DOI: 10.1007/978-3-540-30574-3_19.

[PTT11] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. "Optimal Verification of Operations on Dynamic Sets". In: 2011, pp. 91–110. DOI: 10.1007/978-3-642-22792-9_6.

[PH23] Shahar Papini and Ulrich Haböck. *Improving logarithmic derivative lookups using GKR*. Cryptology ePrint Archive, Paper 2023/1284. 2023. URL: https://eprint.iacr.org/2023/1284.

[PPS21] Chris Peikert, Zachary Pepin, and Chad Sharp. "Vector and Functional Commitments from Lattices". In: 2021, pp. 480–511. DOI: 10.1007/978-3-030-90456-2_16.

[Pol22] Polygon Labs. *Introducing Plonky2*. Accessed: 2025-08-07. Jan. 2022. URL: https://polygon.technology/blog/introducing-plonky2.

[PK22] Jim Posen and Assimakis A. Kattis. *Caulk+: Table-independent lookup arguments*. Cryptology ePrint Archive, Report 2022/957. 2022. URL: https://eprint.iacr.org/2022/957.

[ris25a] risc0 project contributors. *Proof System Sequence Diagram*. https://github.com/risc0/risc0/blob/912c2e198f3abc1094fa55e45840febaee203c22/website/docs/proof-system/proof-system-sequence-diagram.md. Accessed: 2025-10-05. 2025.

[ris25b] risc0 project contributors. *risc0/risc0: RISC Zero — zero-knowledge verifiable computing platform*. https://github.com/risc0/risc0. Accessed: 2025-10-05. 2025.

[Ros+24] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. "Hekaton: Horizontally-Scalable zkSNARKs Via Proof Aggregation". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, 929–940. ISBN: 9798400706363. DOI: 10.1145/3658644.3690282. URL: https://doi.org/10.1145/3658644.3690282.

[SGS20] Gili Schul-Ganz and Gil Segev. "Accumulators in (and Beyond) Generic Groups: Non-trivial Batch Verification Requires Interaction". In: 2020, pp. 77–107. DOI: 10.1007/978-3-030-64378-2_4.

[Sch80] J. T. Schwartz. "Fast Probabilistic Algorithms for Verification of Polynomial Identities". In: *J. ACM* 27.4 (Oct. 1980), 701–717. ISSN: 0004-5411. DOI: 10.1145/322217.322225. URL: https://doi.org/10.1145/322217.322225.

[Set20] Srinath Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: 2020, pp. 704–737. DOI: 10.1007/978-3-030-56877-1_25.

[Set+18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. "Proving the correct execution of concurrent services in zero-knowledge". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, 339–356. ISBN: 9781931971478.

[ST25] Srinath Setty and Justin Thaler. *Twist and Shout: Faster memory checking arguments via one-hot addressing and increments*. Cryptology ePrint Archive, Paper 2025/105. 2025. URL: https://eprint.iacr.org/2025/105.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. *Customizable constraint systems for succinct arguments*. Cryptology ePrint Archive, Paper 2023/552. 2023. URL: https://eprint.iacr.org/2023/552.

[STW24] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. "Unlocking the Lookup Singularity with Lasso". In: 2024, pp. 180–209. DOI: 10.1007/978-3-031-58751-1_7.

[Sou25] Lev Soukhanov. *Logup*: faster, cheaper logup argument for small-table indexed lookups*. Cryptology ePrint Archive, Paper 2025/946. 2025. URL: https://eprint.iacr.org/2025/946.

[Sou24] Lev Soukhanov. *WARPfold : Wrongfield ARithmetic for Protostar folding*. Cryptology ePrint Archive, Paper 2024/354. 2024. URL: https://eprint.iacr.org/2024/354.

[Sri+22] Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. "Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators". In: 2022, pp. 2719–2733. DOI: 10.1145/3548606.3560676.

[Sta25] StarkWare Labs. *Stwo: a next-generation Rust implementation of a CSTARK prover and verifier*. https://github.com/starkware-libs/stwo. Version v1.0.0 released July 18, 2025. July 2025.

[SLZ24] Haochen Sun, Jason Li, and Hongyang Zhang. "zkLLM: Zero Knowledge Proofs for Large Language Models". In: Dec. 2024, pp. 4405–4419. DOI: 10.1145/3658644.3670334.

[Sze+23] Alan Szepieniec, Alexander Lemmens, Jan Ferdinand Sauer, Bobbin Threadbare, and Al-Kindi. *The Tip5 Hash Function for Recursive STARKs*. Cryptology ePrint Archive, Paper 2023/107. 2023. URL: https://eprint.iacr.org/2023/107.

[Tha24] Justin Thaler. *FAQ on Jolt's Initial Implementation*. a16z Crypto (blog post). Apr. 2024. URL: https://a16zcrypto.com/posts/article/faqs-on-jolts-initial-implementation/.

[Tha13] Justin Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation". In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 71–89. ISBN: 978-3-642-40084-1.

[Tom+20] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. "Aggregatable Subvector Commitments for Stateless Cryptocurrencies". In: 2020, pp. 45–64. DOI: 10.1007/978-3-030-57990-6_3.

[Tya+22] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. "VeRSA: Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 2793–2807. ISBN: 9781450394505. DOI: 10.1145/3548606.3560605. URL: https://doi.org/10.1145/3548606.3560605.

[Val08] Paul Valiant. "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency". In: *Proceedings of the 5th Conference on Theory of Cryptography*. TCC'08. New York, USA: Springer-Verlag, 2008, 1–18. ISBN: 354078523X.

[Cry] "Verifiable Computation for Approximate Homomorphic Encryption Schemes". In: Aug. 2025, pp. 643–677. ISBN: 978-3-032-01906-6. DOI: 10.1007/978-3-032-01907-3_21.

[WW23] Hoeteck Wee and David J. Wu. "Succinct Vector, Polynomial, and Functional Commitments from Lattices". In: 2023, pp. 385–416. DOI: 10.1007/978-3-031-30620-4_13.

[Woo+25] Anna Woo, Alex Ozdemir, Chad Sharp, Thomas Pornin, and Paul Grubbs. "Efficient Proofs of Possession for Legacy Signatures". In: May 2025, pp. 3291–3308. DOI: 10.1109/SP61157.2025.00080.

[Xin+25] Zhibo Xing et al. "Zero-Knowledge Proof-Based Verifiable Decentralized Machine Learning in Communication Network: A Comprehensive Survey". In: *IEEE Communications Surveys & Tutorials* (2025), 1–1. ISSN: 2373-745X. DOI: 10.1109/comst.2025.3561657. URL: http://dx.doi.org/10.1109/COMST.2025.3561657.

[Xio+22] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. *VERI-ZEXE: Decentralized Private Computation with Universal Setup*. Cryptology ePrint Archive, Paper 2022/802. 2022. URL: https://eprint.iacr.org/2022/802.

[Zap+22a] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. "Caulk: Lookup Arguments in Sublinear Time". In: 2022, pp. 3121–3134. DOI: 10.1145/3548606.3560646.

[Zap+22b] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. *Baloo: Nearly Optimal Lookup Arguments*. Cryptology ePrint Archive, Report 2022/1565. 2022. URL: https://eprint.iacr.org/2022/1565.

[Zca25] Zcash Developers. *Lookup Argument*. https://zcash.github.io/halo2/design/proving-system/lookup.html. Accessed: 2025-06-23. Zcash, 2025.

[Zha+20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Xiaodong Song. "Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof". In: *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 859–876. URL: https://api.semanticscholar.org/CorpusID:209467198.

[ZSG24] Yuncong Zhang, Shi-Feng Sun, and Dawu Gu. "Efficient KZG-Based Univariate Sum-Check and Lookup Argument". In: *Public-Key Cryptography – PKC 2024: 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, NSW, Australia, April 15–17, 2024, Proceedings, Part II*. Sydney, NSW, Australia: Springer-Verlag, 2024, 400–425. ISBN: 978-3-031-57721-5. DOI: 10.1007/978-3-031-57722-2_13. URL: https://doi.org/10.1007/978-3-031-57722-2_13.

[Zha+23] Yuncong Zhang, Shi-Feng Sun, Ren Zhang, and Dawu Gu. "Polynomial IOPs for Memory Consistency Checks in Zero-Knowledge Virtual Machines". In: *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part II*. Guangzhou, China: Springer-Verlag, 2023, 111–141. ISBN: 978-981-99-8723-8. DOI: 10.1007/978-981-99-8724-5_4. URL: https://doi.org/10.1007/978-981-99-8724-5_4.

[Zha+18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. "vRAM: Faster Verifiable RAM with Program-Independent Preprocessing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 908–925. DOI: 10.1109/SP.2018.00013.

[Zip79] Richard Zippel. "Probabilistic algorithms for sparse polynomials". In: *Symbolic and Algebraic Computation*. Ed. by Edward W. Ng. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 216–226. ISBN: 978-3-540-35128-3.

A Table Size and Structure vs Performance

Whenever the table $\mathbf{t}$ has significantly more elements than the witness size, $|\mathbf{t}|$ can easily yield to a bottleneck in proving performance (see also Figure 1). A class of lookup arguments that are useful for the setting of large tables is that of *table-efficient* lookup arguments[17], in which the running time of the prover grows sublinearly with the tables. Table-efficient arguments can roughly be divided into two families. The first family obtains table-efficiency by relying on some underlying *structure* of the large table. For the sake of this discussion, the reader can intuitively think of structured tables, as those that can be reconstructed in sublinear time relative to their size[18]. In this setting, the family of constructions based on Lasso [STW24; CFR25; ST25] offers the best tradeoffs between proving time and other performance metrics (without relying on recursion).

The second family of schemes supporting large tables does not require any particular structure; instead, it relies on a *preprocessing* stage. The auxiliary material computed at this stage is of size $\Omega(|\mathbf{t}|)$, but the prover will require only a sublinear amount at the proving time. We observe a few patterns in the efficiency profile of this last class of construction (see Fig. 1 for a visual taxonomy). A first set of constructions obtains table-efficiency through the use of specialized proving arguments with *native sublinear proving time* [Cam+25] (e.g., for polynomial evaluation). At the time of writing, these constructions are mostly of theoretical interest since all the currently known instantiations yield very large constants both for preprocessing and proving time.

A second class of constructions—those from the cq family [EFG22; Cam+24; CFG25; ZSG24]—offers highly performant provers, but feature an expensive preprocessing stage. As a consequence, they are mostly applicable to larger tables up to a threshold size, where the preprocessing stage starts becoming infeasible (such a threshold depends on the underlying machine being used).

The constructions that offer the best balance in terms of proving and preprocessing performance are those that obtain lookups through a straightforward approach: use a lightweight accumulator scheme (such as a Merkle Tree) to commit to the table, thus obtaining a digest acc; at preprocessing time, precompute membership proofs for all the elements in the table (e.g., all the internal nodes of the Merkle tree); when proving a lookup, use a general-purpose SNARK to prove the statement "I know $\mathbf{w}$ and membership proofs $\boldsymbol{\pi}$ such that π_i certifies the membership of $\mathbf{w}_i$ with respect to acc". We refer to these constructions as "non-black-box" in that they require explicitly encoding the membership verification as a circuit. Instances of this approach are used, for example, in Pantry [Bra+13].

A.1 Research gaps on large tables

Efficient lookups based on polynomial preprocessing. Can we find instantiations of sublinear polynomial commitment schemes (such as those in [Cam+25]) that are *concretely* efficient?

Other useful structures in tables. Can we find other techniques, outside of the Lasso approach, to exploit some forms of structure in tables?

Non-pairing efficient schemes. The best proving time and proof size are achieved through *pairings* (by the cq family). Can we obtain similar performance from different assumptions?

Lightwight preprocessing. Can we construct black-box solutions (maybe from pairings themselves) with a balanced proving profile (where preprocessing is feasible for larger tables)?

[17] Also occasionally referred to as *super-efficient* lookup arguments [CFG25].
[18] We describe this notion more precisely in Appendix G.3.

B Deferred Definitions

Lemma 1 (Univariate Sumcheck). *Let $\mathbb{H} \subseteq \mathbb{F}$ be a multiplicative subgroup of size t. For any univariate f of degree less than t, the following holds:*

$$\sum_{a \in \mathbb{H}} f(a) \;=\; t \cdot f(0).$$

B.1 Polynomial Commitment Scheme

Definition 10. *[Polynomial Commitment Scheme] A* Polynomial Commitment Scheme *is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *such that:*

- $(\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(\lambda, k)$*: On input the security parameter λ and maximum degree k (number of variables in case a multilinear polynomial), this algorithm outputs a pair of verifier key (vk) and prover key (pk). This algorithm might require randomness as nondeterministic input too.*
- $\mathsf{C} \leftarrow \mathsf{Commit}(\mathsf{pk}, p(X))$*: On input pk and a polynomial $p(X)$, it outputs a commitment C. This algorithm might take randomness as non-deterministic input too, but for simplicity, we don't consider it here.*
- $\pi \leftarrow \mathsf{Open}(\mathsf{pk}, p(X), x)$*: On input pk, $p(X)$, and input x, outputs an evaluation proof π that $y = p(x)$.*
- $1/0 \leftarrow \mathsf{Verify}(\mathsf{vk}, \mathsf{C}, x, \pi, y)$*: On input vk, the commitment C, input x, evaluation y, and the proof of the correct evaluation, it outputs a bit indicating acceptance or rejection.*

A polynomial commitment scheme is said to be secure if it is complete and knowledge-sound as defined below:

Completeness. *It captures the fact that an honest prover will always convince the verifier. Formally, for any efficient adversary $\mathcal{A}$, we have:*

$$\Pr\left[\mathsf{Verify}_{\mathsf{PCS}}(\mathsf{srs}_{\mathsf{PCS}}, \mathsf{C}, x, \pi_{\mathsf{PCS}}, p(x)) = 1 \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}_{\mathsf{PCS}}(\lambda, k), \\ p(X) \leftarrow \mathcal{A}((\mathsf{vk}, \mathsf{pk})), \\ \mathsf{C} \leftarrow \mathsf{Commit}(\mathsf{pk}, p(X)), \\ \pi_{\mathsf{PCS}} \leftarrow \mathsf{Open}(\mathsf{pk}, p(X), x) \end{array}\right] = 1$$

Extractability. *Captures the fact that whenever the prover provides a valid opening, it knows a valid pair $(p(\boldsymbol{X}), y)$, where $p(\boldsymbol{x}) = y$. Formally, for all PPT adversaries $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ such that the probability of the following event is negligible:*

$$\Pr\left[\begin{array}{l} \mathsf{Verify}_{\mathsf{PCS}}(\mathsf{vk}, \mathsf{C}, x, \pi_{\mathsf{PCS}}, y) = 1 \\ \wedge\, p(x) \neq y \end{array} \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}_{\mathsf{PCS}}(\lambda, k), \\ \mathsf{C} \leftarrow \mathcal{A}((\mathsf{vk}, \mathsf{pk})), \\ p(X) \leftarrow \mathcal{E}((\mathsf{vk}, \mathsf{pk}), \mathsf{C}, k), \\ x \leftarrow \mathcal{A}((\mathsf{vk}, \mathsf{pk}), \mathsf{C}), \\ (\mathsf{vk}, \mathsf{pk}), p(X), x) \end{array}\right]$$

KZG polynomial commitment scheme. KZG [KZG10] is a polynomial commitment scheme that forms the foundation of numerous lookup table constructions. Its popularity comes from several key features: it supports batch openings of a single polynomial at multiple points, as well as batch openings of multiple polynomials at multiple points [FK23; Bon+20a]. This efficiency makes KZG a core component in SNARK frameworks such as Plonk [GWC19] and in a wide range of lookup argument protocols [GW20; Zap+22a; EFG22; PK22; Zap+22b; GK22]. KZG enjoys desirable algebraic properties such as homomorphic commitment and fast amortised proofs. A naive opening at a single point costs $O(d)$ time, where d is the degree of the polynomial. However, opening a set of d evaluation points can be achieved in $O(d \log^2 d)$ time. Moreover, if the evaluation points form a subgroup of the underlying finite field, the opening can be performed in quasilinear time $O(d \log d)$ [FK23].

B.2 Argument of knowledge

Definition 11 (Argument of Knowledge). *Consider a relation $\mathcal{R}$ over public parameters, structure, instance, and witness tuples. A non-interactive argument of knowledge for $\mathcal{R}$ consists of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$, denoting the generator, the prover, the verifier and the encoder respectively with the following interface.*

- $\mathcal{G}(1^\lambda) \to \mathsf{pp}$*: On input security parameter λ, samples public parameters* pp.
- $\mathcal{K}(\mathsf{pp}, s) \to (\mathsf{pk}, \mathsf{vk})$*: On input structure s, representing common structure among instances, outputs the prover key* pk *and verifier key* vk.
- $\mathcal{P}(\mathsf{pk}, u, w) \to \pi$*: On input instance u and witness w, outputs a proof π proving that* $(\mathsf{pp}, s, u, w) \in \mathcal{R}$.
- $\mathcal{V}(\mathsf{vk}, u, \pi) \to \{0,1\}$*: On input the verifier key* vk*, instance u, and a proof π, outputs 1 if the instance is accepting and 0 otherwise.*

Completeness. *A non-interactive argument of knowledge satisfies completeness if for any PPT adversary $\mathcal{A}$:*

$$\Pr\left[\mathcal{V}(\mathsf{vk}, u, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (s, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, s, u, w) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, s), \\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, u, w) \end{array}\right] = 1.$$

Knowledge soundness. *A non-interactive argument of knowledge is knowledge soundness if for all PPT adversaries $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that for all randomness ρ:*

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{vk}, u, \pi) = 1, \\ (\mathsf{pp}, s, u, w) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (s, u, \pi) \leftarrow \mathcal{A}(\mathsf{pp}; \rho), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, s), \\ w \leftarrow \mathcal{E}(\mathsf{pp}, \rho) \end{array}\right] = \mathsf{negl}(\lambda).$$

A non-interactive argument of knowledge is succinct if the size of the proof π is polylogarithmic in the size of the statement proven.

B.3 Accumulation scheme

Definition 12. *[Accumulation Scheme] An accumulation scheme [Bün+21; BC23] for a predicate $\phi : X \to \{0,1\}$ is a tuple of algorithms $\Pi_{\mathsf{acc}} = (\mathsf{Setup}_{\mathsf{acc}}, \mathsf{P}_{\mathsf{acc}}, \mathsf{V}_{\mathsf{acc}}, \mathsf{D}_{\mathsf{acc}})$, all of which have access to the same random oracle $\mathsf{O}_{\mathsf{acc}}$, such that:*

- $\mathsf{Setup}_{\mathsf{acc}}(1^\lambda) \to \mathsf{srs}_{\mathsf{acc}}$ *: On input the security parameter, outputs public parameters* $\mathsf{srs}_{\mathsf{acc}}$*. For simplicity, we assume that all functions implicitly take* $\mathsf{srs}_{\mathsf{acc}}$ *as input.*
- $\mathsf{P}_{\mathsf{acc}}\big(\mathsf{st}, \pi, \mathsf{acc}_1\big) \to \big(\mathsf{acc}, \mathsf{pf}\big)$*: The accumulation prover implicitly given* $\mathsf{srs}_{\mathsf{acc}}$*, statement* st*, predicate inputs $\pi = (\pi.x, \pi.w)$, and an accumulator $\mathsf{acc}_1 = (\mathsf{acc}_1.x, \mathsf{acc}_1.w)$, outputs a new accumulator* acc *and corrections terms* pf.
- $\mathsf{V}_{\mathsf{acc}}\big(\mathsf{acc}_1.x, \mathsf{acc}_2.x, \mathsf{pf}\big) \to \mathsf{acc}.x$*: The accumulation verifier implicitly given* $\mathsf{srs}_{\mathsf{acc}}$*, and on input the instances of two accumulators, and the accumulation proof outputs a new accumulator instance* $\mathsf{acc}.x$.
- $\mathsf{D}_{\mathsf{acc}}\big(\mathsf{acc}\big) \to 1/0$*: The decider takes as input* acc *and accepts or rejects.*

and satisfies completeness and soundness as defined below:

Completeness. *For all fresh proofs π such that $\phi(\pi) = 1$ and accumulator* acc *such that* $\mathsf{D}_{\mathsf{acc}}(\mathsf{acc}) = 1$*, the following holds:*

$$\Pr\left[\begin{array}{c} \mathsf{V}_{\mathsf{acc}}\big(\mathsf{acc}_1.x, \mathsf{acc}_2.x, \mathsf{pf}\big) = \mathsf{acc}.x \\ \wedge\ \mathsf{D}_{\mathsf{acc}}\big(\mathsf{acc}\big) = 1 \end{array} \;\middle|\; \begin{array}{r} \mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda) \\ \big(\mathsf{acc}, \mathsf{pf}\big) \leftarrow \mathsf{P}_{\mathsf{acc}}\big(\mathsf{st}, \pi, \mathsf{acc}_1\big) \end{array}\right] = 1$$

Knowledge-soundness. *For every PT adversary $\mathcal{A}$, there exists a polynomial-time extractor $\mathcal{E}$ such that the following probability is negligible:*

$$\Pr\left[\begin{array}{c}(D_{\mathsf{acc}}(\mathsf{acc}_1) \neq 1 \vee\ D_{\mathsf{acc}}(\mathsf{acc}_2) \neq 1) \wedge \\ \mathsf{V}_{\mathsf{acc}}(\mathsf{srs}, \mathsf{acc}_1.x, \mathsf{acc}_2.x, \mathsf{pf}) = \mathsf{acc}.x \\ \wedge\ \mathsf{D}_{\mathsf{acc}}(\mathsf{acc}) = 1\end{array} \middle| \begin{array}{r}\mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{acc}, \mathsf{acc}_1.x, \mathsf{acc}_2.x, \mathsf{pf}) \leftarrow \mathcal{A}(\mathsf{srs}) \\ (\mathsf{acc}_1.w, \mathsf{acc}_2.w) \leftarrow \mathcal{E}(\mathsf{acc}, \mathsf{acc}_1.x, \mathsf{acc}_2.x, \mathsf{pf})\end{array}\right] = 1$$

B.4 Preprocessing PIOP

Definition 13 (Preprocessing PIOP for Lookup Relation [Cam+25]). *Let* $\mathrm{I} := (\mathbb{S}, n, \mathbf{t})$ *be a lookup index and* OLK_I *be the corresponding relation. A PIOP for* OLK_I *is defined by a tuple* $(\mathcal{I}, \mathcal{P}, \mathcal{V})$ *such that:*

- ***Offline phase:*** *$\mathcal{I}(\mathrm{I})$ is a deterministic algorithm that, given the index* I*, generates preprocessing oracle polynomials $p_1, \ldots, p_i$. We denote by $\mathbf{i_x}$ the oracles to these polynomials provided to the verifier, and by $\mathbf{i_w}$ the actual polynomials p_i, as held by the prover. For simplicity, we assume that after this preprocessing step, the verifier and prover respectively retain only their corresponding $\mathbf{i_x}$ and $\mathbf{i_w}$, and no longer require access to* I*.*
- ***Online phase:*** *The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ run an interactive protocol, compactly denoted by $\langle \mathcal{P}(\mathbf{i_w}, \mathbf{w}), \mathcal{V}(\mathbf{i_x}, \mathbf{x}) \rangle$, where in each round the prover $\mathcal{P}$ sends oracle polynomials, and the verifier $\mathcal{V}$ sends random challenges. The verifier can query all the oracles, the ones sent by the prover and the ones generated by $\mathcal{I}$, at arbitrary points. At the end of the execution, $\mathcal{V}$ accepts or rejects.*

Such a PIOP is complete, sound, knowledge-sound and zero-knowledge as defined later. Importantly, we assume that the offline phase is conducted honestly; that is, the oracles generated for p_1, p_2, ..., p_i are generated honestly, whereas the oracles generated in the online phase could have been generated maliciously.

- *Completeness:* For any honest execution of the protocol $\langle \mathcal{P}(\mathbf{i_w}, \mathbf{w}), \mathcal{V} \rangle (\mathbf{i_x}, \mathbf{x})$, where $(\mathrm{I}, \mathbf{x}; \mathbf{w}) \in \mathrm{OLK}_\mathrm{I}$ and $(\mathbf{i_x}, \mathbf{i_w}) \leftarrow \mathcal{I}(\mathrm{I})$, the verifier accepts.
- *δ-knowledge Soundness:* A PIOP is δ-knowledge-sound if there exists a PPT oracle machine $\mathcal{E}$ (which can query all the oracle polynomials at arbitrary points) such that for any adversary $\mathcal{A}$ and any $(\mathbf{i_w}, \mathbf{x})$, the following probability holds:

$$\Pr\left[\langle \mathcal{A}(\mathbf{i_w}, \mathbf{x}), \mathcal{V}(\mathbf{i_x}, \mathbf{x}) \rangle = 1 \wedge (\mathrm{I}, \mathbf{x}; \mathbf{w}) \in \mathrm{OLK}_\mathrm{I} \mid \mathbf{w} \leftarrow \mathcal{E}_\mathcal{A}(\mathbf{i_w}, \mathbf{x})\right] \leq \delta.$$

- *Zero-knowledge:* As in standard definitions of PIOP, such an interaction is said to be zero-knowledge if there exists a simulator that, given the index and the instance, can generate a transcript that is computationally indistinguishable from a real PIOP transcript.

C Decomposability

One key technique for building efficient lookup arguments—independent of the specific lookup protocol but rather a property of the underlying table—is called *decomposability*. This concept was first introduced in Arya [Boo+18] for bitwise operations, and was later pushed to its limits in Lasso [STW24] and Jolt [AST24], which construct lookup arguments for tables so large (e.g., of size 2^{128}) that they cannot even be materialized. At a high level, decomposability is a property of a table $\mathbf{t}$ that allows a lookup of a value v in $\mathbf{t}$ to be reduced to lookups of related values $v_1, v_2, \ldots, v_k$ in smaller subtables $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$. When such a reduction exists, we say that $\mathbf{t}$ can be decomposed into $\{\mathbf{t}_1, \ldots, \mathbf{t}_k\}$. This is particularly useful when the total size of the subtables, $\sum |\mathbf{t}_i|$, is significantly smaller than $|\mathbf{t}|$. A simple example arises in smallness proofs. Suppose we want to prove that a value v is less than 2^{128} over a large enough field.

Directly checking membership in a table of size 2^{128} is infeasible—even if each table entry were just one bit, the table would require billions of terabytes. Instead, we can decompose v into four 32-bit limbs: $v = v_0 + 2^{32} \cdot v_1 + 2^{64} \cdot v_2 + 2^{96} \cdot v_3$ and then check that each limb v_i is less than 2^{32}. This requires only a small lookup table of size 2^{32} per limb. We formalise the notion of decomposability in Definitions 14 and 15.

Definition 14 (Decomposability of Tables). *A table $\mathbf{t} \in \mathbb{S}^N$ is said to be decomposable into tables $\mathbf{t}_i \in \mathbb{S}_i^{N_i}$ for $1 \leq i \leq k$ if there exists a map $\mathcal{M} : \mathbb{S} \to \mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_k$ such that:*

$$s \in \mathbb{S} \ : \ \mathcal{M}(s) = (s_1, s_2, \ldots, s_k) \implies$$

$$s \in \mathbf{t} \quad \iff \quad s_i \in \mathbf{t}_i \quad \textit{for all } 1 \leq i \leq k.$$

Note that the tables $\mathbf{t}_i$ are not necessarily distinct; for instance, they may all be identical, as in the smallness test described above. The definition above can also be extended to indexed tables (see Definition 15). We naturally extend the definition of the map $\mathcal{M}_{\text{set}} : \mathbb{S} \to \mathbb{S}_1 \times \cdots \times \mathbb{S}_k$ to tuples by defining $\mathcal{M}_{\text{set}} : \mathbb{S}^r \to \mathbb{S}_1^r \times \cdots \times \mathbb{S}_k^r$, where the map is applied component-wise to each entry of the input tuple.

Definition 15 (Decomposability of Indexed Tables). *An indexed table $\mathbf{t} \in \mathbb{S}^N$ is said to be decomposable into indexed tables $\mathbf{t}_i \in \mathbb{S}_i^{N_i}$ for $1 \leq i \leq k$ if there exists maps*

$$\mathcal{M}_{set} : \mathbb{S} \to \mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_k \quad \textit{and} \quad \mathcal{M}_{index} : [N] \to [N_1] \times [N_2] \times \cdots \times [N_k],$$

such that:

$$\forall\, s \in \mathbb{S},\, j \in [N] \ : \ \mathcal{M}_{set}(s) = (s_1, s_2, \ldots, s_k),\, \mathcal{M}_{index}(j) \to (j_1, j_2, \ldots, j_k) \implies$$

$$s = \mathbf{t}[j] \quad \iff \quad s_i = \mathbf{t}_i[j_i] \quad \textit{for all } 1 \leq i \leq k.$$

How to concretely apply decomposition? Let us first review the lookup procedure. The client holds a commitment C to the witness vector $\mathbf{w}$ and wants to verify that $\mathbf{w} \subseteq \mathbf{t}$, where $\mathbf{t}$ is a public lookup table. To leverage decomposability, the vector $\mathbf{w}$ is transformed via the mapping $\mathcal{M}_{\text{set}}$ into k vectors $(\mathbf{w}_1, \ldots, \mathbf{w}_k)$. This mapping is applied entry-wise. The prover must now demonstrate two properties for each subvector $\mathbf{w}_i$:

1. *Lookup relation.* That $\mathbf{w}_i \subseteq \mathbf{t}_i$, where $\mathbf{t}_i$ is the corresponding public table for $\mathbf{w}_i$. This is typically done via a lookup proof π_i. For many tables of interest [AST24], tables $\mathbf{t}_i$ are the same.[19] Now, if the lookup argument supports homomorphic proofs [EFG22], all the proofs π_i can be aggregated and verified together.
2. *Correctness of decomposition.* The subvectors $(\mathbf{w}_1, \ldots, \mathbf{w}_k)$ are indeed the result of applying the map $\mathcal{M}_{\text{set}}$ to $\mathbf{w}$. In other words, the commitments $(C_1, \ldots, C_k)$ to the subvectors must be consistent with the original commitment C under the map $\mathcal{M}_{\text{set}}$.

The verifier only receives the commitments C_i (without the corresponding openings). Therefore, the prover must demonstrate that these commitments open to vectors whose entries satisfy the following relation, under the entrywise application of the map $\mathcal{M}_{\text{set}}(\cdot)$:

$$\mathcal{M}_{\text{set}}(\mathbf{w}) = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$$

This decomposition check itself amounts to a *commit-and-prove* statement. If verifying this relation requires recomputing commitments within an arithmetic circuit, it may undermine the efficiency benefits that lookup arguments are designed to provide. Jolt observes that for lookup tables of interest—such as those used for CPU operations—the corresponding mappings $\mathcal{M}_{\text{set}}$ and $\mathcal{M}_{\text{index}}$ are *linear* functions. For example, in a smallness check where a 128-bit value

[19] Even if the tables are identical, k separate lookups are still required in general, since given commitments C_i to vectors $\mathbf{w}_i$, in most settings, it is non-trivial to compute a commitment to the concatenation of these vectors $\mathbf{w}_1 || \mathbf{w}_2 || \ldots || \mathbf{w}_k$.

is decomposed into four 32-bit chunks (v_0, v_1, v_2, v_3), the relation is also linear: $v = v_0 + 2^{32} \cdot v_1 + 2^{64} \cdot v_2 + 2^{96} \cdot v_3$. If the underling commitment scheme supports linear homomorphisms, this relation lifts directly to commitments:

$$C = C_0 + 2^{32} \cdot C_1 + 2^{64} \cdot C_2 + 2^{96} \cdot C_3.$$

This allows verifying the decomposition without opening any commitments. Even if the commitment scheme is not homomorphic, it is still straightforward to evaluate such linear relations non-homomorphically. For example, if a PCS is used to commit to the polynomials (vectors) $\mathbf{w}_i$ to obtain C_i, and the prover provides the openings of the C_i's at a random point r (which the verifier can check individually using the PCS verification), then it is sufficient for the verifier to check that:

$$\mathbf{w}(r) = \mathbf{w}_0(r) + 2^{32} \cdot \mathbf{w}_1(r) + 2^{64} \cdot \mathbf{w}_2(r) + 2^{96} \cdot \mathbf{w}_3(r).$$

According to the Schwartz-Zippel lemma [DL78; Zip79; Sch80], this relation holds over the entire domain with high probability if it holds at a randomly chosen point r, assuming the degree of the polynomials is bounded. This allows taking advantage of decomposable tables with non-homomorphic commitments, such as the hash-based ones.

D Existing Approaches: Multiset Equality Technique

D.1 Arya

Arya [Boo+18] was the first work to formally define a lookup relation (unindexed) to simplify constraint systems. Let the table values be denoted by $\{t_i\}_{i\in[N]}$, and the witness values to be looked up by $\{w_i\}_{i\in[n]}$. Arya defines polynomials $T(X)$ and $W(X)$, whose roots correspond to these sets, as follows:

$$W(X) := \prod_{i\in[n]} (X - w_i), \qquad T(X) := \prod_{i\in[N]} (X - t_i).$$

The lookup relation is equivalent to showing that these two polynomials share the same root set, disregarding multiplicities. Formally, this amounts to demonstrating the existence of non-negative integers $\{m_i\}_{i\in[N]}$ such that

$$W(X) = \prod_{i\in[N]} (X - t_i)^{m_i}. \tag{1}$$

The prover commits to polynomials $T(X)$ and $W(X)$, and then demonstrates correctness of Equation (1) given a challenge β sampled by the verifier:

$$\prod_i (\beta - w_i) \;=\; \prod_i (\beta - t_i)^{m_i}.$$

This verification proceeds through a square-and-multiply approach: the prover commits to the multiplicities m_i by committing to their binary representations. The resulting protocol is relatively involved and mainly of theoretical interest, but it achieves good asymptotic behaviour in the broader context of proving RAM computations. A simpler and more practical protocol, inspired by Arya, was later introduced in Plookup [GW20], which we describe next.

D.2 Plookup

Plookup [GW20] is a follow-up to Arya, the goal of Plookup is to prove that a multiset $\mathbf{w} := \{w_1, w_2, \ldots, w_n\}$ is contained in a table set $\mathbf{t} := \{t_1, t_2, \ldots, t_N\}$. Without loss of generality, one may assume that every element of the table also appears in the witness; this can be ensured by defining the witness as $\mathbf{w} \cup \mathbf{t}$. The key observation is that if the witness multiset $\mathbf{w}$ is sorted

relative to $\mathbf{t}$, then $\mathbf{w} \subseteq \mathbf{t}$ if and only if their successive difference sets (not *multisets*) coincide, up to zero repetitions, i.e.:

$$\{w_2 - w_1, w_3 - w_2, \ldots, w_n - w_{n-1}\} = \{t_2 - t_1, \ldots, t_N - t_{N-1}\} \cup \{0\}.$$

Witness being sorted can be enforced via a permutation PIOP similar to the permutation PIOP described in Plonk [GWC19]. Given the observation above on difference sets, Plookup introduces a PIOP for the lookup relation, constructed using univariate polynomials. While their initial instantiation relies on KZG, the construction is generic and applies to any univariate PCS, and even works over smaller finite fields [DCB25]. The approach in Plookup is similar to the reordering-based read-write memory in vRAM [Zha+18], with some simplifications arising from the unindexed lookup argument setting. Another subsequent work, i.e. HyperPlonk [Che+23] builds on Plookup, designing an equivalent lookup argument in the multilinear setting, with the same underlying technique and characteristics.

D.3 Halo2

Halo2 [Zca25] constructs a lookup PIOP inspired by offline memory checking protocols, which reduces a global subset check into local constraints. Consider a witness vector $\mathbf{w} \in \mathbb{F}^n$ and a table vector $\mathbf{t} \in \mathbb{F}^n$, and suppose we want to prove that $\mathbf{w} \subseteq \mathbf{t}$. If the witness and table have different sizes, we can pad the table with arbitrary elements and the witness with elements from the table. To transform this global condition into local constraints, we first permute $\{w_i\}$ into a new array $\{w'_i\}$ such that equal values are grouped together and aligned with corresponding values in a permuted version of the table $\{t'_i\}$. We then enforce the following local constraint:

$$(w'_i - t'_i) \cdot (w'_i - w'_{i-1}) = 0.$$

This condition guarantees that each entry in the permuted witness either matches a table element (the first factor is zero) or is equal to the previous witness entry (the second factor is zero), thereby ensuring the subset relation is satisfied locally.

E Existing Approaches: Accumulator-Based

One approach to constructing lookup tables is through the use of *set accumulators* that support *batch commit-and-prove* openings. Batching is necessary because we want to generate a single proof that an entire subtable is contained within the full table. The *commit-and-prove* aspect arises from the fact that, unlike standard accumulators, the verifier only receives a commitment to the subtable—not the subtable itself. At a high level, the idea is that in the preprocessing, the verifier commits to the full table $\mathbf{t}$ and the prover additionally generates opening proofs for each element. These precomputed proofs can later be used to efficiently generate a single proof about $\mathbf{t}' \subseteq \mathbf{t}$, for any committed subtable $\mathbf{t}'$. Importantly, the time to compute this proof depends only on the size of the subtable, not on the size of the entire table. When using accumulators, the table and the subtables are treated and committed as sets. Therefore, to obtain a full-fledged lookup argument, this approach requires additionally proving that each witness' entry $\mathbf{w}_i \in \mathbf{t}'$, where $\mathbf{w}$ is a committed *vector*.

E.1 Flookup

Flookup [GK22] follows the accumulators approach, combined with prior approaches inspired by Caulk and Caulk+. Specifically, Flookup commits to the table and the subtables, modeled as sets, using a classical pairing-based accumulator [Ngu05], and then provides protocols to prove the correctness of the subtable extraction and that the entries of the committed witness are in the subtable.

E.2 Duplex

Starting from accumulator-based techniques, Duplex [Han+25] builds a zero-knowledge lookup argument where the table is committed as an element in groups of unknown orders. This setting includes RSA groups as well as (in its transparent instantiation) class groups [BH11]. In contrast to the bilinear pairing regime, groups of unknown order offer public parameters of constant size. In order to support a witness committed through Pedersen commitments in a *prime order* group—and since most SNARKs do not natively support commitments in unknown order groups—the main technical challenge solved by [Cam+22a; Han+25] is to provide techniques to efficiently link accumulators with commit-and-prove SNARKs and without encoding RSA-style operations inside a circuit. While the approach in HARiSA [Cam+22a] explicitly obtains a proof system for batch set membership (and may not guarantee security in the presence of duplicate elements, Duplex extends its techniques to the lookup setting (where the witness may contain duplicates).

F Existing Approaches: Logup-Based

Lemma 2 (Logup [Eag+24; Hab22]). *Let $\mathbb{F}$ be a field of characteristic $p > \max(n, N)$. Given two sequences of field elements $\{w_i\}_{i=1}^n$ and $\{t_i\}_{i=1}^N$, we have $\{w_i\} \subseteq \{t_i\}$ as sets (with multiplicities ignored) if and only if there exists a sequence $\{m_i\}_{i=1}^N$ of field elements such that:*

$$\sum_{i=1}^{n} \frac{1}{x + w_i} = \sum_{i=1}^{N} \frac{m_i}{x + t_i}.$$

We stress the importance of the characteristic of the underlying field, which must satisfy $p > \max(n, N)$. This requirement immediately rules out fields with very small characteristic, such as binary fields (characteristic 2). Nevertheless, it is still possible to work over relatively small prime fields as long as their size exceeds both N and n. A concrete example is the *BabyBear* field, defined as the prime field $\mathbb{F}_p$ with $p = 2^{31} - 1$ [HLP24]. This field is already used in Stwo zkVM [Sta25], where its lookup argument (Logup) operates over the BabyBear field. Logup lemma is also extended to a vectorised version as described below.

Lemma 3 (Vectorized Logup [Hab22]). *Let $\mathbb{F}$ be a field of characteristic $p > \max(n, N)$, and let $v \in \mathbb{N}$ denote the vector length. Suppose we are given two sets of vectors $\{\mathbf{w}_i\}_{i=1}^n \subseteq \mathbb{F}^v$ and $\{\mathbf{t}_i\}_{i=1}^N \subseteq \mathbb{F}^v$. Define the polynomials $w_i(y) := \sum_{j=1}^v w_{i,j} \cdot y^{j-1}$ and $t_i(Y) := \sum_{j=1}^v t_{i,j} \cdot y^{j-1}$ where $\mathbf{w}_i = (w_{i,1}, \ldots, w_{i,v})$ and $\mathbf{t}_i = (t_{i,1}, \ldots, t_{i,v})$. Then, the set inclusion $\{\mathbf{w}_i\}_{i=1}^n \subseteq \{\mathbf{t}_i\}_{i=1}^N$ (ignoring multiplicities) holds if and only if there exists a sequence of field elements $\{m_i\}_{i=1}^N$ such that:*

$$\sum_{i=1}^{n} \frac{1}{x + w_i(y)} = \sum_{i=1}^{N} \frac{m_i}{x + t_i(y)}.$$

As described in Section 3.4, an indexed lookup can be constructed from a vector lookup by representing the table $\mathbf{t}$ as a collection of pairs (i, t_i) and applying the vector lookup to this representation [BC24; Sou25]. Next, we present a modification of the Logup lemma that enables a projective variant. This is achieved by introducing a binary selection vector $\mathbf{s}$, where each entry s_i is set to 1 if the corresponding value w_i is intended to be looked up in the table, and 0 otherwise.

Lemma 4 (Projective Logup). *Let $\mathbb{F}$ be a field of characteristic $p > \max(n, N)$. Let $\{w_i\}_{i=1}^n$ and $\{t_i\}_{i=1}^N$ be two sequences of elements in $\mathbb{F}$. Define a selector vector $\mathbf{s} = \{s_i\}_{i=1}^n$ with $s_i \in \{0, 1\}$ indicating whether w_i is selected for lookup. Then, the set of selected values $\{w_i \mid s_i = 1\}$ is a subset of $\{t_i\}_{i=1}^N$ (as sets, ignoring multiplicities) if and only if there exists a sequence $\{m_i\}_{i=1}^N$ of field elements such that:*

$$\sum_{i=1}^{n} \frac{s_i}{x + w_i} = \sum_{i=1}^{N} \frac{m_i}{x + t_i}.$$

Proof. It directly reduces to the original logup lemma:

$$\sum_{i=1}^{n} \frac{s_i}{x + w_i} = \sum_{i\,:\,s_i \neq 0} \frac{1}{x + w_i} = \sum_{i=1}^{N} \frac{m_i}{x + t_i}.$$

F.1 Logup+GKR

To provide background, GKR protocol [GKR15], and its subsequent improvements [CMT12; Tha13], are arguments of knowledge for layered arithmetic circuits. They are particularly well-suited for *uniform* circuits—intuitively, where each layer admits a simple polynomial description that can be preprocessed. A notable property of GKR is that it requires commitments only to the input and output layers, without committing to the intermediate wires. This reduces the number of commitments, albeit at the cost of performing more field operations. However, since field operations are significantly cheaper than group operations, this trade-off results in a more efficient overall prover cost. The limitation is that the circuit must have a simple layered structure. An interesting application arises when combining GKR with the Logup lemma [PH23; BC24], which checks that the sum of derivatives on both sides of the Logup equation are equal. Given commitments to both the lookup vector and the table, and an additional commitment to a multiplicity vector, one can prove that the Logup equality holds —with no extra commitments. In this setting, GKR can be used as follows: the circuit receives all numerators and denominators of the involved fractions, and at each layer, pairs and sums the fractions, halving their number at each layer in a binary tree structure. This regular structure enables the efficient application of GKR. A similar technique was used in Proofs for Deep Thoughts [BC24], where it was applied in the context of an accumulation scheme and IVC, and extended to accumulate the GKR verifier itself. Another recent work [Dor24] explores decomposable lookup tables in the GKR+Logup setting.

In the GKR+Logup setting, on the negative side, the GKR circuit depends on the size of the lookup table, as it requires all numerators and denominators of the fractions to be provided as inputs. On the positive side, this technique does not rely on homomorphism and is therefore compatible with hash-based commitments. It has already been applied in practice in zkVMs such as Stwo [Sta25].

F.2 cq (cached quotients)

cq [EFG22] is the state-of-the-art for *super-sublinear* lookup tables. cq first reduces the lookup statement into an equality of rational function sums using the Logup lemma. Through a preprocessing step based on the batch KZG opening technique of Feist and Khovratovich [FK23], this approach achieves a fast prover time of $O(n \log n)$, which is independent of the table size. More precisely, translating $\{w_i\}_{i\in[n]} \subseteq \{t_i\}_{i\in[N]}$ via the Logup lemma yields:

$$\sum_{i\in[N]} \frac{m_i}{x + t_i} = \sum_{i\in[n]} \frac{1}{x + w_i} \tag{2}$$

Replacing the variable x with the verifier challenge α, each side of this equation is then interpolated as a polynomial. Specifically:

- The left-hand side is interpolated over a subgroup $\Omega_1 = \{\omega^i\}_{i\in[N]}$ of size N as a polynomial p_1 of degree at most N, i.e., $p_1(\omega^i) = \frac{m_i}{\alpha + t_i}$.
- The right-hand side is interpolated over a subgroup $\Omega_2 = \{\omega^i\}_{i\in[n]}$ of size n, resulting in a polynomial p_2 of degree at most n, i.e., $p_2(\omega^i) = \frac{1}{\alpha + w_i}$.

Given that KZG is a homomorphic scheme and the fact that committing to the zero polynomial is essentially *free*, the prover can commit to both sides of the equation efficiently, i.e. linear in n and independent of N. Then, using a univariate sumcheck protocol [BS+19], the prover and verifier can jointly check whether the identity above holds. However, it is not enough to

verify the equality alone; the prover must also demonstrate that the committed polynomials p_1 and p_2 were constructed correctly. Checking that p_2 is correctly formed is straightforward. The verifier already has a polynomial commitment interpolating witness $w(x)$ over a subgroup Ω_2 of size n, and can verify for all $\omega \in \Omega_2$ that $p_2(\omega) = (\alpha + w(\omega))^{-1}$ by existing standard techniques. Checking that p_1 is well-formed amounts to proving the existence of a quotient polynomial $q(X)$ such that:

$$\forall\, \omega \in \Omega_1 : \quad p_1(\omega) \cdot (t(\omega) + \alpha) - m(\omega) = q(\omega) \cdot z_{\Omega_1}(\omega),$$

where $t(X)$ is the interpolating polynomial over the table, $m(X)$ is the interpolated numerator of the rational expression, i.e., vector of multiplicities m_i, α is the verifier's challenge, and $z_{\Omega_1}(X)$ is the vanishing polynomial over the subgroup Ω_1 of size N. This is equivalent to showing:

$$\forall\, \omega \in \Omega_1 : \quad p_1(\omega) \cdot t(\omega) = q(\omega) \cdot z_{\Omega_1}(\omega) + r(\omega),$$

for some remainder $r(X)$ of degree less than N. If the verifier is given KZG commitments to each of these polynomials, using a pairing check, it can verify the identity above. The polynomials p_1 and r are already sparse on Ω_1, so the prover can compute their commitments independently of N. The commitments to t and z_{Ω_1} are precomputed, so it only remains to compute the commitment to the polynomial q, which is not necessarily sparse. The novelty of the paper is that computing a commitment to the polynomial q can be preprocessed by computing commitments to some cached quotients such that computing the commitment to $q(X)$ only requires $O(n)$ operations and is independent of the table size.

Prior to cq [Zap+22a; PK22; GK22; Zap+22b], KZG-based lookup arguments had a prover time that included a term of $O(n \log^2 n)$, where n is the size of the lookup witness. This overhead was due to the need to compute the quotient subtable explicitly. As a result, all four prior works were forced to perform polynomial interpolation and evaluation over arbitrary sets, rather than over multiplicative subgroups. Algorithms operating over arbitrary sets generally have $O(n \log^2 n)$ complexity, in contrast to the $O(n \log n)$ achievable on structured domains like subgroups. In contrast, cq avoids this bottleneck by reducing the lookup argument to a sum of rational functions via the Logup lemma. This approach eliminates the need to extract any quotient subtable, leading to a more efficient prover runtime.

Projective cq. Using Lemma 4, with a small modification, cq can support projective lookups. This time, the polynomial p_2, which interpolates the terms of the sum $\sum_{i\in[n]} \frac{1}{x+w_i}$, is replaced with a polynomial that interpolates $\sum_{i\in[n]} \frac{s_i}{x+w_i}$. Given a public polynomial $s(X)$ that interpolates the vector $\mathbf{s} = \{s_i\}_{i\in[n]}$ on Ω_2, checking that p_2 is well-formatted is equivalent to checking

$$s(\omega) \cdot (\alpha + w(\omega))^{-1} \;=\; p_2(\omega), \quad \forall\, \omega \in \Omega_2,$$

which can be done using standard polynomial checks. The rest of the protocol remains unchanged. The only additional cost of this adjustment is that the prover also requires opening the polynomial $s(X)$ of degree n.

Multilinear cq. The original cq construction is based on KZG, a univariate polynomial commitment scheme, which is not natively compatible with multilinear SNARKs such as HyperPlonk. Campanelli, Fiore, and Gennaro [CFG25] observed that in Equation (2), only the left-hand side relies on KZG—specifically to compute commitments to cached quotients during preprocessing—while the witness $\mathbf{w}$ on the right-hand side polynomial p_2 can instead be committed via a multilinear polynomial commitment. In this case, verifying that p_2 is well-formed is replaced with polynomial checks suitable for the multilinear setting. The only check requiring both sides of the equation is the equality check, which originally verified $\sum_{\omega\in\Omega_1} p_1(\omega) = \sum_{\omega\in\Omega_2} p_2(\omega)$, using a univariate sumcheck for both sides. In the modified version, the right-hand side is handled using the standard multilinear sumcheck, while the left-hand side remains unchanged. This modification allows using cq with multilinear SNARKs that commit to the witness via a multilinear polynomial.

Other extensions to cq. Locq [ZSG24] is a related work that optimises cq, achieving improvements in proof size and verification cost by replacing cq's univariate sumcheck protocol [BS+19] with a new sumcheck protocol, at the expense of requiring additional trusted setups beyond the KZG setup. The work in [Cam+24] proposes several efficiency improvements to cq:

- cq+, an overall improvement over cq compared to which it reduces communications size. This proof system, in turn, admits two additional variants, described below.
- cq++, which reduces even further the proof size of cq+ trading it against a modest additional overhead for the verifier (one additional pairing).
- zkcq+, a variant of cq+ with *full* zero-knowledge, that is it hides both the table and the witness.

Both cq+ and cq++ admit a zero-knowledge version (achieving hiding of the witness, but not of the table) with minor overhead in proof size. An overview of these variants can be seen in Table 2.

Scheme	Preprocessing	Proof Size	Proving Cost	Verifier Cost	Zero-knowledge
cq [EFG22]	$O(N \log N)$	$8\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$5P$	×
cq+ [Cam+24]	$O(N \log N)$	$7\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$5P$	×
cq++ [Cam+24]	$O(N \log N)$	$6\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$6P$	×
cq+(zk) [Cam+24]	$O(N \log N)$	$8\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$5P$	✓
cq++(zk) [Cam+24]	$O(N \log N)$	$7\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$6P$	✓
zkcq+ [Cam+24]	$O(N \log N)$	$9\mathbb{G}_1$	$8mG_1 + O(m \log m)\mathbb{F}$	$6P$	✓✓
Locq [ZSG24]	$O(N \log N)$	$4\mathbb{G}_1 + 1\mathbb{G}_2$	$6mG_1 + mG_2 + O(m \log m)\mathbb{F}$	$4P$	✓✓

Table 2: An overview of the efficiency of different cq variants. P, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{F}$ denote pairing operation, group operation on $\mathbb{G}_1$, group operations on $\mathbb{G}_2$ and field operation respectively. In the zero-knowledge column, a single ✓ denotes that the proof hides only the witness but not the table, while a double ✓✓ indicates that the proof hides both the table and the variant.

G Existing Approaches: Subtable Extraction

G.1 Caulk and Caulk+

Caulk [Zap+22a] introduced the first sublinear lookup argument, achieving a prover complexity of $O(n^2 + n \log N)$, which was later improved by subsequent work Caulk+ [PK22], to a complexity of $O(n^2)$. These schemes satisfy a property called *position-hiding linkability*, meaning that the proof doesn't reveal any information about the elements within the witness.

The central idea of Caulk and Caulk+ is to construct a commitment to the polynomial corresponding to subtable $\mathbf{t}_I$ that includes all table elements that appear in the witness. By computing the commitment to $t_I(X)$ efficiently through the subvector aggregation techniques of Tomescu et al. [Tom+20], the prover's cost becomes sublinear to the size of the original table and instead scales only with $|\mathbf{t}_I|$, which is bounded by the size of the witness. To demonstrate that $t_I(X)$ is a polynomial corresponding to a valid subtable, the prover must establish the following identity:

$$t(X) - t_I(X) = z_I(X)q_I(X),$$

where $t(X)$ is the polynomial interpolating the full table, $t_I(X)$ is the polynomial interpolating the subtable, $z_I(X)$ is the vanishing polynomial over the subtable indices I that has to be kept secret, and $q_I(X)$ is the quotient polynomial witnessing that $t(X)$ and $t_I(X)$ agree on all points in I. In both Caulk and Caulk+, commitment to the quotient polynomial $q_I(X)$ is computed via a linear combination of preprocessed commitments, which correspond to subtables that exactly exclude one element.

Due to the position-hiding property, one of the main challenges is to convince the verifier that $z_I(X)$ vanishes over the right roots of unity without revealing I, this is where Caulk and Caulk+ differ. In a high-level, Caulk designs an arithmetic circuit to prove $z_I(X)$ is well-formatted. We will not enter into the details, as Caulk+ greatly simplifies and improves upon this. To ensure the correct computation of $z_I(X)$, Caulk+ reuses the preprocessing idea from Tomescu et al. and proceeds as follows. Suppose the table is interpolated as a polynomial over a group Ω. The prover first demonstrates that the vanishing polynomial z_I vanishes over a subset of Ω. Equivalently, this means proving the existence of a quotient polynomial $Q(X)$ such that

$$z_I(X) \cdot Q(X) = z_\Omega(X) = X^N - 1.$$

An attentive reader may observe that one drawback of this approach is that computing $Q(X)$ can require $O(N)$ time. To address this, the prover relies on precomputations and aggregation techniques, as described in [Tom+20], to build the quotient efficiently. Next, Caulk+ applies a mapping that sends the indices I into a subgroup of order $|I|$, allowing the verifier to compute its vanishing polynomial efficiently.

G.2 Baloo

Baloo [Zap+22b] builds directly upon the matrix–vector technique introduced at the beginning of this section. Similar to Caulk and Caulk+, Baloo extracts an intermediate subtable $\mathbf{t}_I$, which contains precisely those table elements that appear in the witness. By extracting $\mathbf{t}_I$ efficiently using preprocessed polynomials, Baloo ensures that the prover's work remains independent of the size of the original table. After extracting the subtable $\mathbf{t}_I$, the prover must show that there exists an elementary matrix M such that $M \times \mathbf{t}_I = \mathbf{w}$. Rather than proving this relation directly, inspired by prior work such as Marlin [Chi+20], the verifier samples a random vector $\mathbf{r}$ and, with overwhelming probability, reduces the proof to verifying the scalar relation $(\mathbf{r} \times M) \cdot \mathbf{t}_I = \mathbf{r} \cdot \mathbf{w}$, thereby simplifying the original matrix–vector equation.

Remark 1. The main novelty of cq compared to Caulk, Caulk+, and Baloo lies in its use of Logup rather than explicitly extracting the subtable $\mathbf{t}_I$. Logup enables a direct connection between the original table and the lookup witness, making the prover's work completely independent of the table size.

G.3 Lasso

Lasso [STW24] designs a lookup PIOP over *structured tables*—tables whose multilinear extensions can be evaluated in sublinear time relative to their size. Beyond structured tables, Lasso also supports *decomposable tables*, enabling efficient lookups in massive tables of size 2^{256} that cannot even be materialized. In contrast to KZG-based schemes, Lasso requires no additional setup beyond that of the underlying PCS. Although its prover cost scales linearly with the (decomposed) table size, it remains practically efficient. Since Lasso employs a sparse PCS to commit to the map matrix, it retains efficiency even when instantiated with non-homomorphic PCS, such as the hash-based schemes, as demonstrated in Binius [DP25]. To get an understanding of Lasso, we first need to review Spark.

Spark [Set20]. Spark is a *sparse* multilinear polynomial commitment scheme in which both the commitment and opening times depend only on the number of non-zero evaluation points, and are independent of the size of the evaluation domain. It was introduced in the context of the Spartan PIOP for R1CS, where, during setup, an *honest* indexer commits to the multilinear extensions of the sparse matrices A, B, and C of the R1CS. Later, in the online phase, the prover opens these sparse matrices at random evaluation points. In this setting, the matrices have size $O(n^2)$ but contain only $O(n)$ non-zero elements. For both the indexer's setup phase and the prover's online phase, it is therefore crucial that committing to and opening the polynomial can be done in $O(n)$ time. For a polynomial f with N variables, its evaluation at a point $\mathbf{x}$ can be written as:

$$f(\mathbf{x}) = \sum_{\mathbf{w}\in\{0,1\}^N} f(\mathbf{w}) \cdot \mathsf{eq}(\mathbf{x},\mathbf{w}) = \sum_{\mathbf{w}\in\{0,1\}^N : f(\mathbf{w})\neq 0} f(\mathbf{w}) \cdot \mathsf{eq}(\mathbf{x},\mathbf{w}) \tag{3}$$

The core idea of Spark is to commit to the positions of the non-zero elements. For instance, a sparse matrix can be uniquely represented by the set of tuples $\{(\mathrm{row}_i, \mathrm{col}_i, \mathrm{val}_i)\}$ corresponding to its non-zero entries. During opening, the prover computes Equation (3) inside the arithmetic circuit. Although the summation therein involves only n terms, evaluating each $\mathsf{eq}(\mathbf{x},\mathbf{w})$ requires $O(\log N)$ constraints. A direct encoding would therefore incur a constraint count of $O(n \log N)$. To reduce this cost, Spark exploit the tensor-product structure of eq and uses *online-constructed* lookup tables. Specifically, observe that:

$$\mathbf{x}_1, \mathbf{x}_2, \mathbf{w}_1, \mathbf{w}_2 \in \{0,1\}^{\frac{N}{2}} \implies \mathsf{eq}(\mathbf{x}_1 \| \mathbf{x}_2,\ \mathbf{w}_1 \| \mathbf{w}_2) = \mathsf{eq}(\mathbf{x}_1, \mathbf{w}_1) \cdot \mathsf{eq}(\mathbf{x}_2, \mathbf{w}_2).$$

Given an evaluation point $\mathbf{x} \in \{0,1\}^{\log N}$, split it into $\mathbf{x}_1, \mathbf{x}_2 \in \{0,1\}^{\log \frac{N}{2}}$. The prover using offline-memory checking techniques such as Spice [Set+18], constructs two tables:

$$\mathcal{T}_1 = \{\mathsf{eq}(\mathbf{x}_1, \mathbf{w}) \,:\, \mathbf{w} \in \{0,1\}^{\log \frac{N}{2}}\} \quad \text{and} \quad \mathcal{T}_2 = \{\mathsf{eq}(\mathbf{x}_2, \mathbf{w}) \,:\, \mathbf{w} \in \{0,1\}^{\log \frac{N}{2}}\}.$$

By splitting the input vectors, the computation of $\mathsf{eq}(\mathbf{x}_1\|\mathbf{x}_2, \mathbf{w}_1\|\mathbf{w}_2)$ can be reduced to two independent lookups—one in $\mathcal{T}_1$ and one in $\mathcal{T}_2$. Each table has size $2^{\frac{N}{2}}$, in contrast to the 2^N entries required for a direct table containing all $\mathsf{eq}(\mathbf{x},\cdot)$ values. This idea generalizes naturally: for $\mathbf{x} \in \{0,1\}^{\log N}$, we can partition $\mathbf{x}$ into c segments $\mathbf{x}_1, \ldots, \mathbf{x}_c$, each of length $\log \frac{N}{c}$, and build c separate tables, each of size $2^{\frac{N}{c}}$. Specifically, by choosing c such that $N = c \cdot n$, the prover only needs to construct a constant number of tables of size n and perform $c \cdot n$ lookups in the tables. This guarantees that the overall complexity of the prover remains $O(n)$.

In Spartan, Spark is applied to commit to public sparse matrices during the setup phase. Since these matrices are public, it is naturally assumed that the commitment to the sparse polynomial is performed honestly, and the scheme is not zero-knowledge. In contrast, Lasso employs Spark within its lookup PIOP during the online phase. As discussed in Section 3.2, polynomials committed online by the prover may be committed maliciously. Lasso extends the analysis of Spark and shows that the original scheme, without modification, also supports the case of maliciously committed polynomials. Consequently, it becomes the first standard polynomial commitment scheme for sparse multilinear polynomials with such security guarantees.

However, this scheme still does not provide zero-knowledge, and therefore, Lasso lookups are also not zero-knowledge, which was later addressed by Campanelli, Faonio, and Russo [CFR25].

How Lasso works based on Spark? Lasso models the lookup relation as a matrix-vector multiplication $M_{n\times N} \times \mathbf{t}_{N\times 1} = \mathbf{w}_{n\times 1}$ where M is an elementary matrix. Considering the multilinear extension of the above terms, this identity turns out to be equivalent, up to negligible soundness error, to verifying that for a random $\mathbf{r} \in \{0,1\}^{\log n}$:

$$\sum_{\mathbf{y}\in\{0,1\}^{\log N}} \widetilde{M}(\mathbf{r},\mathbf{y}) \cdot \widetilde{\mathbf{t}}(\mathbf{y}) \;=\; \widetilde{\mathbf{w}}(\mathbf{r}) \tag{4}$$

Now, for a given table and witness $\mathbf{t}$ and $\mathbf{w}$, the prover must prove the following compound statement: the prover knows an elementary matrix M such that Equation (4) holds. Firstly, to prove that Equation (4), by the sumcheck protocol [Lun+92], the summation can be reduced to evaluations of $\widetilde{M}(\mathbf{r}_1,\mathbf{r}_2)$, $\widetilde{\mathbf{t}}(\mathbf{r}_1)$, and $\widetilde{\mathbf{w}}(\mathbf{r}_2)$ at random points $\mathbf{r}_1 \in \{0,1\}^{\log N}$ and $\mathbf{r}_2 \in \{0,1\}^{\log n}$. Here, $\widetilde{\mathbf{w}}$ is committed via a dense polynomial commitment and therefore the prover can provide the verifier with the opening. The table $\widetilde{\mathbf{t}}$ is structured so that the verifier can evaluate it directly, leaving only the multilinear extension $\widetilde{M}$. The prover commits to the sparse multilinear extension $\widetilde{M}$, i.e. using Spark, the prover commits to the dense polynomials $\{(\mathrm{row}_i, \mathrm{col}_i, \mathrm{val}_i)\}$ which specify the positions and values of non-zero matrix entries. For Lasso, all non-zero entries are 1, so values need not be committed. Since row and column indices lie in $[0,n)$ and $[0,N)$, the prover only commits to small field elements—critical in group-based commitment schemes, where committing to small elements is roughly $10\times$ cheaper than to random elements. The prover then opens the polynomial at the given point using the Spark opening.

Finally to prove that M is an elementary matrix—each row containing exactly one 1—it suffices, under the representation $\{(\mathrm{row}_i, \mathrm{col}_i, \mathrm{val}_i)\}$, to verify that the set of row indices is exactly $[0,n)$. This ensures every row has exactly one non-zero entry, and explicit commitment to row indices is unnecessary.

Generalized Lasso. The original work introduces a variant of Lasso, referred to as the *generalized Lasso*. In this setting, the table $\mathbf{t}$ may not be decomposable, although its multilinear extension remains efficiently computable. Non-decomposability can increase the size of the polynomials in the sumcheck Equation 4, particularly the map matrix M. To address this, Lasso employs a *sparse sumcheck protocol*, which ensures that the prover avoids computation over zero entries. After the sumcheck, the remainder of the protocol closely follows standard Lasso. The lookup relation is reduced to random evaluations of $\widetilde{M}$, $\widetilde{\mathbf{t}}$, and $\widetilde{\mathbf{w}}$, where the prover commits to $\widetilde{M}$ using Spark and to $\widetilde{\mathbf{w}}$ using a dense polynomial commitment, allowing them to open these polynomials at a random point for the verifier. Finally, $\widetilde{\mathbf{t}}$ can be efficiently evaluated by the verifier itself, since we assumed its multilinear extension is efficiently computable.

Projective Lasso. It is straightforward to generalize the matrix–vector form of the lookup relation to also support *projective lookups*. In this setting, for witness entries that are *not* intended to be looked up, the corresponding row in the matrix consists entirely of zeros. We allow the matrix M to contain only two types of rows: *elementary rows*—zero everywhere except for a single entry equal to 1 and *all-zero* rows. In Lasso's current formulation, since each row is assumed to contain exactly one non-zero entry, the protocol does not commit to the vector $\{\mathrm{row}_i\}$ of non-zero positions—this set is assumed to be exactly equal to $[0,n)$. Under our generalization, however, this vector *must* also be committed to. This commitment can be performed during the setup phase, as the indices of the witness entries to be looked up are already known during setup. The rest of the protocol remains unchanged. We emphasize that this approach has not appeared in prior work. For example, in order to avoid projective lookup, Jolt commits to witness in each step of the VM separately and checks the correctness of lookups via explicit bookkeeping within the circuit.

Other related work. Lasso is the core component of Jolt [AST24]. At a high level, Jolt is a companion paper demonstrating that it is possible to replace all the instruction set in RISC-5 with decomposable and structured lookups. By substituting RISC-5 instructions with Lasso

lookups, Jolt constructs an efficient and open-source zkVM [a1625]. As mentioned earlier, Spark employs an offline memory-checking technique to implement online lookup tables for the eq function. However, the offline memory-checking technique is incompatible with binary fields (fields of characteristic two), as it requires the field characteristic to be at least as large as the number of lookups. Binius [DP25], a hash-based SNARK over a binary field, demonstrates how to adapt the offline memory-checking technique so that Lasso becomes compatible with their proof system. A related work is FLI [GM24], which aims to make Jolt compatible with recursive proof systems. To this end, they design an *accumulation scheme* for Lasso lookups. We elaborate more in Section H.3. The work in [CFR25] provides a zero-knowledge version of Lasso and shows how to apply it to obtain zkVM proofs that are non-malleable.

G.4 Shout

Shout [ST25] is a family of lookup[20] PIOPs that build upon the ideas of Generalized Lasso. A key observation is that, although the map matrix M in Lasso consists only of 0s and 1s, Spark requires committing to several random values per lookup due to its underlying offline memory checking. This introduces a prover bottleneck both concretely and asymptotically.

Shout introduces a parameter d. In the simplest setting, Shout-1 is essentially the Generalized-Lasso protocol without using the Spark sparse polynomial commitment scheme. Instead, the prover commits to the map matrix. This can be done efficiently using a standard elliptic-curve-based PCS such as Hyrax or KZH, since committing to 0's and 1's is 2–3 orders of magnitude cheaper than committing to random elements. In this setting, the prover only needs to commit to n number of 1s, where n denotes the number of lookups. However, elliptic-curve-based schemes such as KZH require a setup that depends on the *degree* of the polynomial rather than its sparsity. As a result, they are practical only for small tables, while large tables become infeasible due to the linear-sized setup. On the other hand, with typical hash-based commitments, committing to zeros is not free—it is concretely expensive to commit to a large, sparse matrix.

Shout observes that the tensor structure used in Spark for eq functions can be applied directly to the map matrix. Each row of the matrix is a basis vector—i.e., a vector that is zero everywhere except for a single entry equal to 1. These basis vectors possess a natural tensor structure and can be expressed as a tensor product of smaller vectors. For $d > 1$, Shout further exploits that every *one-hot* vector (a vector in $\{0,1\}^K$ with exactly one entry equal to 1) can be decomposed into a tensor product of d smaller one-hot vectors, each of length $K^{1/d}$. This decomposition helps control the negative effects of a large matrix map, in particular, the size of the commitment setup when using curve-based PCS and the total number of committed bits when using hashing-based PCS. Shout-d therefore decomposes each basis vector into d smaller vectors. The prover now commits to $(d-1) \cdot N^{1/d}$ zero values and d ones per 1 entry in the original matrix. This comes at the cost of higher-complexity constraints in the resulting systems, which increase from rank-1 to rank-d.

H Existing Approaches: Lookup Accumulation

Incremental Verifiable Computation (IVC). IVC [Val08] is an argument of knowledge that allows proving an iterative computation defined by a function F. IVC is a fundamental primitive with many applications, including verifiable virtual machines [Nex24], succinct blockchains [Bon+20b], verifiable key directories [Tya+22; Haf+25], light clients [Che+20], and more. IVC is generalized by the notion of *Proof-Carrying Data* (PCD) [CT10], which supports verifying distributed computations over a directed acyclic graph (DAG) of computations, as opposed to just a straight-line sequence. Traditional constructions of IVC and PCD relied on recursive SNARKs [Bit+13], encoding the SNARK verifier circuit inside itself to achieve recursion. However, this approach was concretely expensive, since the SNARK verifier circuit is relatively large and inefficient. More recent works [KST22; Bün+21; BC23; Kad+25] introduced

[20] The original paper mainly refers to lookup arguments as *read-only* offline memory checking.

the use of *accumulation (folding)* schemes to construct IVC and PCD. These constructions achieve significantly lower recursive overhead and are much more efficient in practice, making IVC and PCD more viable for real-world applications.

Accumulation (folding) scheme. An *accumulation scheme* [KST22; Bün+21; BC23; Kad+25] (see Definition 12) is an interactive protocol between a prover and a verifier that reduces satisfiability of two NP statements into a single, *accumulated* NP statement. In other words, instead of verifying each statement separately, the verifier only needs to check the validity of the accumulated statement. We say an accumulation scheme is *non-trivial* if the cost of verifying the accumulated statement plus the cost of the accumulation verifier is strictly less than the cost of verifying the two original statements independently. This notion ensures that the verifier gains efficiency by using accumulation, as opposed to simply performing two separate NP verifications. As discussed earlier, accumulation schemes can serve as a primitive to build efficient IVC/PCD schemes. In particular, the BCLMS compiler [Bün+21] shows that, given an accumulation scheme for a specific predicate, one can construct an IVC or PCD scheme for that predicate.

Scheme	Accumulator Prover	Accumulator Verifier	Accumulator Decider	Assumption
Protostar	$O(n)\,\mathbb{G}$	$O(1)\,\mathbb{F}$, $O(1)\,\mathcal{H}$, $3\,\mathbb{G}$	$O(N)\mathbb{G}$	HVC
nLookup	$O(N)\,\mathbb{F}$	$O(n \log N)\,\mathbb{F}$, $\log(n)\,\mathcal{H}$	$O(N)\,\mathbb{F}$	—
FLI	$O(n)\,\mathbb{G}$, $O(n)\,\mathbb{F}$	$O(1)\,\mathbb{F}$, $O(1)\,\mathcal{H}$, $4\,\mathbb{G}$	$O(N \cdot n)\,\mathbb{G}$	HVC

Table 3: Comparison of accumulator costs across three schemes: Protostar, HyperNova, and FLI. The accumulator prover denotes the cost incurred by the prover at each step of IVC. The accumulator verifier represents the recursive verification overhead that must be embedded within the circuit. The accumulator decider corresponds to the cost incurred by the IVC verifier, typically at the final step. "HVC" stands for *homomorphic vector commitment*. We use $\mathbb{F}$ to denote field operations, $\mathbb{G}$ for group operations (e.g., scalar multiplication), and $\mathcal{H}$ for cryptographic hash evaluations. One important observation is that the FLI accumulator prover requires $O(n)$ group operations. Still, it commits only to *small witnesses*, which is approximately $10\times$ more efficient than committing to random elements, which is the case for Protostar.

H.1 Protostar

Protostar [BC23] is a general-purpose compiler that constructs a generic accumulation scheme for any *special-sound protocol*. At a high level, a special-sound protocol is an interactive protocol that admits a special-sound extractor. These protocols are composable: the composition of two special-sound protocols yields another special-sound protocol. Protostar observes that well-known NP languages, such as R1CS and its generalization CCS [STW23], can be expressed as special-sound protocols. Protostar first shows that a lookup relation, as defined in Definition 1, can be transformed into a special-sound protocol using the Logup lemma. Consequently, their general compiler can be applied to construct an accumulation scheme for such lookup relations. To build an IVC scheme for an R1CS relation where the whole witness is required to appear in a public table, Protostar constructs a composition of two special-sound protocols: one for the R1CS relation, one for the lookup check. By the composability property, this combined protocol is also special-sound. Protostar's compiler then produces an accumulation scheme for this composite protocol, which can be compiled into an IVC scheme via the BCLMS compiler. However, as discussed in Section 3.3, we are typically interested in *projective lookup*—that is, verifying that a subset of the witness is included in a public table. We observe that the projective variant of the Logup lemma (Lemma 4) can also be formulated as a special-sound protocol. By composing this protocol in place of the standard lookup protocol, Protostar naturally supports projective lookups. We defer the full details to Appendix I. Proof of Deep Thoughts [BC24] is a

subsequent work that extends Protostar by generalizing its lookup scheme—essentially a read-only memory abstraction—to support write operations as well. As a result, it constructs an accumulation scheme for a more expressive memory model that includes both reads and writes. There are a few important technical points about Protostar that merit further discussion.

Protostar lookup is only efficient for IVC and not PCD. The cost of the Protostar prover per IVC step is linear in the number of lookups and independent of the size of the table. However, the Protostar lookup accumulation is efficient only for constructing IVC schemes, and not for PCD: in their analysis in Section 4.3 of Protostar, the authors explicitly leverage the fact that they are accumulating a fresh (non-relaxed) special-sound lookup protocol with a relaxed one. This is the key reason why the prover's cost per step remains independent of the table size, i.e. building PCD with the same approach, the prover time is not necessarily independent of the table size. The same limitation also applies to Proofs of Deep Thoughts: the scheme is efficient for IVC but not for PCD.

Special-sound protocol composition. We have previously discussed two general strategies for composing relations and lookups—namely, the commit-and-prove approach and PIOP-level composition—the composition method used here is that of *special-sound protocol composition*. We focus on PIOPs among the broader class of interactive protocols because most modern proof systems follow the PIOP+PCS paradigm. However, Protostar takes a different route: it first translates the relation into a special-sound protocol and then constructs an accumulation scheme for it. Thus, the composition of the relation with the lookup check is achieved through the composition of their corresponding special-sound protocols.

Support of gigantic tables. Protostar requires a Pedersen commitment setup proportional to the size of the table, and the runtime of the accumulator decider (i.e., the IVC verifier) is also linear in the table size, which may be problematic for large tables. However, FLI [GM24] demonstrates that Protostar can be extended to support the accumulation of decomposable tables, as introduced in Lasso. This is achieved by reducing inclusion in a large table to inclusion in several smaller tables and then accumulating the inclusion proofs for the smaller tables.

H.2 nLookup (HyperNova)

HyperNova [KS24] introduces a simple and *accumulation-friendly* lookup PIOP over a finite field $\mathbb{F}$. At a high level, consider a lookup table $\mathbf{t}$ of size $N = 2^k$, which we interpret as a function $\mathbf{t} : \{0,1\}^k \to \mathbb{F}$. This table can be uniquely extended to its multilinear extension $\widetilde{\mathbf{t}}(x_1, \ldots, x_k)$. To perform m indexed lookups into this table, we are given m pairs $\{(\mathbf{q}_i, v_i)\}_{i\in[m]}$, where each $\mathbf{q}_i \in \{0,1\}^k$ is a Boolean vector (an index into the table), and the value $v_i \in \mathbb{F}$ is claimed to be $\widetilde{\mathbf{t}}(\mathbf{q}_i)$. The goal is to prove these lookup claims efficiently within a recursive proof system. To do this, HyperNova uses a sumcheck-based folding scheme that reduces the problem of checking m evaluations to checking a single evaluation at a random point. A notable difference in this scheme—although it may not perfectly fit our definition of a committed lookup—is that the verifier (implemented inside the circuit) does not receive a commitment to the looked-up subtable. Instead, all entries are revealed in plaintext, similar to Definition 1. This design choice enables an efficient extension to projective lookups, in particular, certain elements of the witness can be selectively checked without committing to them, an approach adopted in Reef [Ang+24]. Another advantage of this technique is that, unlike FLI and Protostar, which rely on Pedersen commitments, it does not require a large prime field. The protocol costs are as follows:

Verifier's cost (recursive overhead). It includes the sumcheck verifier and consists of $O(\log N)$ field and hash operations, along with $O(m \log N)$ field operations required to compute certain eq polynomials used in the sumcheck protocol. The nLookup protocol implicitly performs a smallness test by providing the indices $\mathbf{q}_i$ as a Boolean vector, which ensures that these indices lie within the range defined by the table size. However, this means that the prover needs to enforce all entries of $\mathbf{q}_i$ are Boolean points. Consequently, such a lookup table may

offer limited savings for simple range-proof tables, as a naive bit-decomposition approach would still require $O(m \log N)$ constraints in the circuit.

Prover's cost (per step). It involves a linear number of field operations to carry out the sumcheck prover.

Decider's cost. As for the accumulation decider (IVC verifier), it must evaluate $\widetilde{\mathbf{t}}(\mathbf{r})$ at a random point $\mathbf{r} \in \mathbb{F}^k$, which in general requires $O(2^k)$ field operations. This cost may be reduced if the table $\mathbf{t}$ exhibits additional structure. For example, if $\widetilde{\mathbf{t}}$ admits an efficiently computable multilinear extension, then the evaluation of $\widetilde{\mathbf{t}}(\mathbf{r})$ can be performed more efficiently than by naive interpolation.

H.3 FLI

IVC (PCD) plays a crucial role in the *continuation* process for zk-VMs [Tha24]. This process involves the prover dividing the program into smaller chunks, proving each one separately, and then aggregating the individual proofs into a single proof. With the motivation of making Jolt *continuation-friendly*, FLI [GM24] builds an accumulation scheme for Lasso lookups. Consider a lookup table $\mathbf{t} \in \mathbb{F}^N$ and a witness vector $\mathbf{w} \in \mathbb{F}^n$. The lookup relation can be represented as a matrix-vector product: $M \cdot \mathbf{t} = \mathbf{w}$, where $M \in \mathbb{F}^{n \times N}$ is an *elementary matrix*—that is, each row of M contains exactly one entry equal to 1, with all other entries being 0. The lookup relation in Lasso can be formalized as follows. Given an index $\mathrm{I} = (\mathbb{F}, n, \mathbf{t})$, the *lookup instance* consists of commitments $\mathrm{com}(\mathbf{t})$, $\mathrm{com}(\mathbf{w})$ and $\mathrm{com}(M)$. The *lookup witness* includes the plaintext matrix M and the vector $\mathbf{w}$, satisfying two constraints: (I) $M \cdot \mathbf{t} = \mathbf{w}$, (II) M is an elementary matrix. A key observation is that the first constraint is *linear*:

$$\begin{cases} M_1 \cdot \mathbf{t} = \mathbf{w}_1 \\ M_2 \cdot \mathbf{t} = \mathbf{w}_2 \end{cases} \implies \forall\, \alpha \in \mathbb{F} : (M_1 + \alpha \cdot M_2) \cdot \mathbf{t} = \mathbf{w}_1 + \alpha \cdot \mathbf{w}_2.$$

Thus, given a *homomorphic commitment scheme* for matrices, these linear constraints can be *accumulated* via linear combinations without introducing error terms. The second constraint—that M is an elementary matrix—can be enforced algebraically through the identities:

$$M \cdot M = M \quad \text{and} \quad M \cdot \mathbf{I} = \mathbf{I} \text{ where } \mathbf{I} = (1\,1\,\ldots\,1)$$

which are reminiscent of R1CS-style constraints. Using techniques inspired by Nova [KST22], FLI builds an accumulation scheme that can handle such constraints efficiently. Finally, FLI extends its accumulation approach to *decomposable tables* by decomposing large tables into smaller *base tables*, allowing lookups to be accumulated across these smaller components as well.

Requirement of homomorphic commitments. As discussed above and similar to Protostar, the commitment to the matrix $M \in \mathbb{F}^{n \times N}$—which contains $n \cdot N$ entries—must be homomorphic. Using commitment schemes with a linear setup, such as Pederson, may impose restrictions on the size of the table N.

Non-efficient decider time. Accumulating sparse matrices via linear combinations poses a challenge: given two matrices M_1 and M_2 with n_1 and n_2 non-zero entries, respectively, their sum may have up to $n_1 + n_2$ non-zero entries. As a result, the accumulated matrix M gradually loses its sparsity over multiple rounds of accumulation. This sparsity loss impacts the decider. Specifically, the decider must recompute $\mathrm{com}(M)$ from the plaintext matrix M. However, since M may contain up to $\min\{\sum n_i,\ |M|\}$ non-zero entries, the decider's workload increases, potentially making the scheme impractical for large matrices. Nevertheless, we believe that this limitation does not hinder the main motivation of the paper, which is to construct an accumulation scheme for Jolt-style lookups. In Jolt, instruction tables are decomposed into relatively small lookup tables of size 2^{16} [a1624]. Therefore, even with densification of M, the decider cost remains manageable in this context.

I Projective Protostar Lookup Accumulation

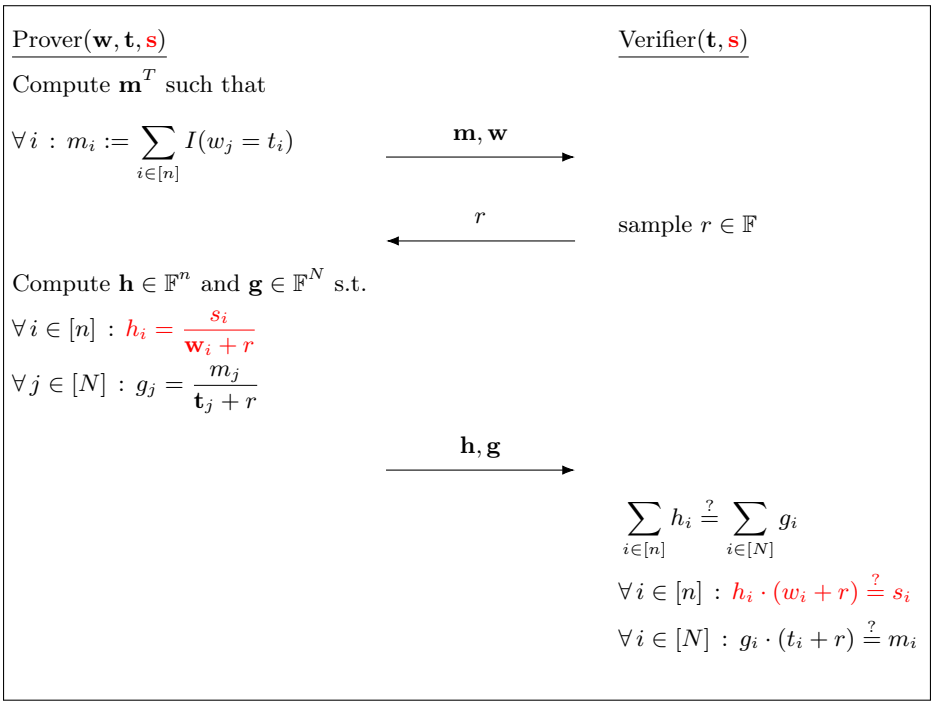

Fig. 2: Special-sound protocol for projective logup

In Figure 2, we present the special-sound protocol for the projective logup. The terms highlighted in red indicate modifications from the original protocol. We assume that both the prover and verifier already have access to the projective indices $\mathbf{s}$, as these are typically generated during circuit preprocessing. This protocol is adapted from the version with negligible completeness error, which suffers from undefined prover messages when $w_i + r = 0$ or $t_j + r = 0$ for some index. Following the suggestion in Protostar, the verifier's checks can be modified to achieve perfect completeness:

$$\big(h_i \cdot (w_i + r) - s_i\big) \cdot (w_i + r) = 0, \quad \big(g_i \cdot (t_i + r) - m_i\big) \cdot (t_i + r) = 0.$$

A proof of special-soundness is left to the reader.