國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

現代零知識查找論證的比較性能分析

A Comparative Performance Analysis of Modern Zero-Knowledge Lookup Arguments

游祖鈞

Tzu-Chun Yu

指導教授: 廖世偉 博士

Advisor: Shih-Wei Liao Ph.D.

中華民國 114 年 7 月

July, 2025

doi:10.6342/NTU202503318

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

現代零知識證明查找論證的比較性能分析

A Comparative Performance Analysis of Modern Zero-Knowledge Lookup Arguments

本論文係游祖鈞君（學號 R11922194）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 114 年 07 月 27 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 27 July 2025 have examined a Master's thesis entitled above presented by TZU-CHUN YU (student ID: R11922194) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

廖世偉		貢敬酵
（指導教授 Advisor）		
蘇中才	鄭振牟	盧暐山

系主任/所長 Director: 陳祝嵩

Acknowledgements

終於寫完這篇論文了，有很多的感謝估計沒有機會親口表達，寫在這裡或許更多是我自己的沉澱。感謝我的教授廖世偉給我很多自由讓我能夠在 SuDo 實習，接觸到真正的區塊鏈，也因此促成機會能夠得到以太坊基金會的 ZK Grant Round 贊助。這個論文的實現必須感謝 Harry, 靖傑, 育銘, 如果沒有一開始你們一起參與 pylookup 的實現，這個 Rust 版本根本不可能完成。也感謝 Plonkish 框架的製作者 Han，基於這框架 Lookup argument 實現變得容易。謝謝 SuDo 的每一個同事，每天生活真的像 CJ 講的一樣過得快樂又一起賺錢，在這裡我真正瞭解區塊鏈，以及鏈上各種 Degen 操作。謝謝以太坊基金會 PSE team 的 Phini, Adrian, Mo, 有你們才有 Acceleration Program 讓我可以碰到 Lookup argument 以及很多很棒的人。特別感謝元宇宙扶輪社頒發給我獎學金，這筆獎學金幫助了我出國參加會議的費用，讓我得以把這個研究作得更好。感謝 Exponential Venture 四巨頭品豪 CharmingJack 你們是真心的好朋友。謝謝我的弟弟妹妹在 24 年暑假我們在巴黎一起跟爸爸度過最後的時光，希望我們永遠都好。最後感謝最重要的媽媽以及小元，你們的陪伴讓我在工作學校與雜七雜八的事情之間有了歇息，希望你們健康快樂，你們是我永遠的後盾。這兩年，加上休學的一年是三年，只能用白駒過隙來形容，每一天都過得很充實，進來資工所之前我有很多想像，雖然最後好像不是按照想像走，但我覺得我更瞭解自己、更追隨內心、也對無謂的社會框架怯魅，我知道我在乎什麼也不在乎什麼。謝謝各位，我們繼續衝鋒

doi:10.6342/NTU202503318

摘要

零知識證明 (Zero-knowledge proofs, ZKP) 是區塊鏈擴展性與隱私保護的關鍵技術，尤其在 ZK rollup 解決方案中扮演核心角色，其透過將大量計算轉移至鏈下，有效提升交易吞吐量。然而，在這些系統中證明複雜的運算（例如虛擬機器的單個操作碼）仍然是主要的性能瓶頸。為此，查找論證 (Lookup Argument) 已成為一項關鍵的最佳化技術，它允許計算步驟的有效性可以根據預定義的表格進行高效驗證，從而避免了成本高昂的算術化過程。

儘管現存多種查找論證協定——包含 Plookup、Caulk、Baloo、CQ、Lasso 與 LogupGKR——各自具備不同的理論複雜度，但市場上始終缺乏一份全面性的實證效能比較，以指導開發者在實際應用中的選擇。

本論文對該領域作出四項關鍵貢獻：首先，我們增強並擴展了一個統一的 Rust 基準測試框架，提供多線性與單變數多項式版本，為未來查找論證研究奠定標準化基礎。其次，我們對六個主流協議進行了廣泛的基準測試，系統性地評估了在不同表格大小與查找密度下的證明者時間、驗證者時間、證明大小及預處理成本。第三，我們發現並解釋了 Lasso 的證明大小在 $K = 12$ 時反直覺地減少的現象，揭示了均勻 Limb 分解 (uniform limb decomposition) 能在多項式承諾方案中實現更高效的批次處理。第四，我們確定了最佳的混合表格查找策略：小表格應使用 LogUp GKR，而大表格則受益於 Lasso 的分解方法。

研究結果從實證角度驗證了查找論證的技術演化路徑：從 Plookup 對表格大

doi:10.6342/NTU202503318

小的線性依賴 ($O(N)$)，到 Caulk 雖然解決了前者問題卻引入了查找數量的平方級瓶頸 ($O(n^2)$)，再到 Baloo 與 CQ 成功將其提升至準線性效率。更重要的是，本研究揭示了 Lasso 與 LogupGKR 等現代協議實現了性能上的典範移轉 (paradigm shift)，其證明者時間不僅比前代協議快上數個數量級，且在測試範圍內幾乎不受表格大小與查找數量的影響。

本論文的結論指出，最佳查找協議的選擇並非絕對，而是一個高度依賴於應用場景的工程決策，涉及在證明者時間、驗證成本、證明大小與預處理開銷之間的多維度權衡。我們提供的實證數據，成功地彌合了漸進理論與現實性能之間的鴻溝，為下一代零知識證明系統的開發者提供了關鍵且實用的選型指南。

關鍵字：查表論證、零知識簡潔非交互式知識論證、多項式承諾

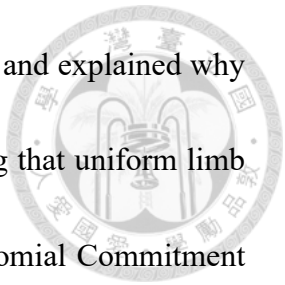
doi:10.6342/NTU202503318

Abstract

Zero-knowledge proofs (ZKPs) are foundational to blockchain scalability and privacy, particularly in ZK-rollups, which enhance transaction throughput by offloading computation from the main chain. However, proving complex operations within these systems, such as individual virtual machine opcodes, remains a significant performance bottleneck. Lookup arguments have emerged as a critical optimization, enabling the efficient verification of computational steps against pre-defined tables, thereby avoiding costly arithmetization. While a proliferation of lookup protocols—including Plookup, Caulk, Baloo, CQ, Lasso, and LogupGKR—offer diverse theoretical complexities, a comprehensive empirical comparison to guide practical implementation has been lacking.

This thesis makes four key contributions to the field: First, we enhanced and extended a unified Rust-based benchmarking framework that provides both multilinear and univariate polynomial versions, creating a standardized foundation for future lookup argument research. Second, we conducted an extensive benchmark of six prominent protocols, systematically evaluating prover time, verifier time, proof size, and preprocessing costs

doi:10.6342/NTU202503318



under varying table sizes and lookup densities. Third, we discovered and explained why Lasso's proof size counter-intuitively decreases at $K = 12$, revealing that uniform limb decomposition enables more efficient batch processing in the Polynomial Commitment Scheme. Fourth, we identified an optimal hybrid table lookup strategy where small tables should use LogUp GKR while large tables benefit from Lasso's decomposition method.

Our results empirically validate the theoretical evolution of these protocols, charting the progression from Plookup's table-size dependency ($O(N)$) and Caulk's lookup-count bottleneck ($O(n^2)$) to the quasi-linear efficiency of Baloo and CQ. Furthermore, we demonstrate that modern protocols like Lasso and LogupGKR achieve a paradigm shift in performance, offering prover times that are orders of magnitude faster and largely independent of table and lookup size within the tested ranges. This study concludes that the optimal choice of a lookup protocol is a highly context-dependent engineering decision, involving trade-offs between prover time, verification cost, proof size, and preprocessing overhead. The empirical data herein provides a crucial, practical guide for developers, bridging the gap between asymptotic theory and real-world performance to inform protocol selection in next-generation ZK-based systems.

Keywords: Lookup Argument, zk-SNARKs, Polynomial Commitment Schemes

doi:10.6342/NTU202503318

Contents

doi:10.6342/NTU202503318

doi:10.6342/NTU202503318

doi:10.6342/NTU202503318

doi:10.6342/NTU202503318

doi:10.6342/NTU202503318

doi:10.6342/NTU202503318

List of Figures

doi:10.6342/NTU202503318

List of Tables

doi:10.6342/NTU202503318

Chapter 1 Introduction

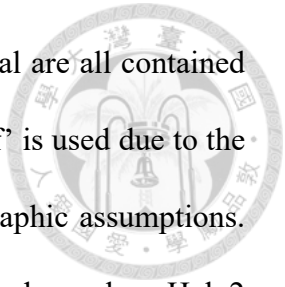
1.1 Research Introduction

Zero-knowledge proofs (referred to as ZKPs hereafter) [9] play an increasingly essential role in the blockchain ecosystem, especially within Ethereum. Their main uses include boosting both privacy and scalability. Concerning privacy, ZKPs allow for transaction validation without disclosing sensitive information. In terms of scalability, they can offload computations from the main chain—a method known as 'rollups' [10]. ZK rollups, categorized as a Layer 2 scaling solution, consolidate several transactions and present a ZKP to the main chain for validity verification, thereby enhancing transaction speed and minimizing costs.

However, proving computational statements within a ZK rollup often involves converting problems into arithmetic circuits, which can be complex and resource-intensive. Early efforts focused on proving Ethereum Virtual Machine execution (ZK-EVM) [11]. Due to challenges, the research community has shifted towards proving compiled versions of Ethereum nodes (ZK-VM), such as those based on RISC-V [12], which offer simplicity and wider adoption.

To further simplify proving, lookup arguments have emerged as a promising tech-

doi:10.6342/NTU202503318



nique [1]. These arguments prove that values of a witness polynomial are all contained within a pre-defined public table. Here, 'argument' rather than 'proof' is used due to the computational soundness of these systems, which relies on cryptographic assumptions. Lookup arguments are now finding their way into production-level code, such as Halo2 [13], and are being employed in the latest ZK-VM designs to prove individual opcode executions.

This paper aims to delve into the application of Lookup Arguments in ZKPs, specifically benchmarking the performance differences among various protocols such as Plookup [1], Baloo [3], CQ [4], and LogupGKR [5]. While theoretical analyses provide insights into the asymptotic complexities of these protocols, real-world performance can deviate significantly. Therefore, we conduct extensive benchmarking to provide a practical evaluation of Lookup Arguments, determining their actual performance characteristics and trade-offs beyond what can be inferred from theoretical analysis alone. This work is motivated by the understanding that theoretical superiority does not always translate to practical efficiency. We aim to demonstrate the true performance characteristics of Lookup Arguments experimentally. We intend to use experimental data to compare their performance under different N/n ratios and analyze the trade-offs in preprocessing, proving time, verification time, and proof size. These analyses will provide a reference for developers when choosing appropriate Lookup protocols and further the advancement of zero-knowledge proof technology in the blockchain field.

doi:10.6342/NTU202503318

1.2 Research Contributions

This thesis makes the following key contributions to the field of applied cryptography and zero-knowledge proof systems:

1. Enhanced Unified Benchmarking Framework: We extended and implemented an existing benchmarking framework to support comprehensive evaluation of lookup argument protocols. Our implementation provides both multilinear and univariate polynomial versions, creating a standardized foundation that future researchers can directly reference when implementing new lookup argument schemes.

2. Comprehensive Empirical Evaluation of Modern Protocols: We conducted an extensive benchmark of six prominent lookup protocols: Plookup [1], Caulk [2], Baloo [3], CQ [4], Lasso [6], and LogupGKR [5]. The evaluation systematically quantifies their performance across four critical metrics: prover time, verifier time, proof size, and preprocessing cost, under a wide range of table sizes and lookup densities.

3. Analysis of Lasso's Table Decomposition Optimization: We discovered and explained why Lasso's proof size counter-intuitively decreases at $K = 12$. The key insight is that uniform limb decomposition (when K is divisible by limb size) enables more efficient batch processing in the Polynomial Commitment Scheme compared to non-uniform decomposition, resulting in smaller proofs despite larger table sizes.

4. Hybrid Table Lookup Strategy: We identified an optimal approach where small tables should use LogUp GKR while large tables benefit from Lasso's decomposition method. Our analysis shows that other existing protocols lack significant advantages in either regime, making this hybrid approach the most efficient overall strategy.

doi:10.6342/NTU202503318

1.3 Research Motivation

The proliferation of zero-knowledge (ZK) rollups [10] has established lookup arguments as a critical optimization for blockchain scalability. These arguments dramatically reduce the proving overhead of complex computations by verifying them against pre-computed tables. This has led to a rapid evolution of protocols, from the foundational Plookup [1] to advanced sublinear-N systems like Caulk [2], Baloo [3], CQ [4], and the paradigm-shifting Lasso [6] and LogupGKR [5].

However, the theoretical complexity of these protocols, often expressed in Big O notation, provides an incomplete picture of their real-world performance. Factors such as constant overheads, implementation-specific optimizations, underlying library efficiency, and practical trade-offs between prover time, verifier time, and proof size are not captured by asymptotic analysis. Developers and researchers currently lack a comprehensive, empirical benchmark that directly compares these systems under unified conditions. This gap between theory and practice creates uncertainty when selecting the most appropriate protocol for a given application, potentially leading to suboptimal engineering decisions. This thesis is motivated by the critical need to bridge this gap by providing a rigorous, practical, and comparative performance analysis of modern lookup arguments.

While state-of-the-art protocols like Lasso [6] and LogupGKR [5] theoretically outperform their predecessors, a narrow focus on only the 'best' systems provides an incomplete picture. A key motivation for this thesis is to move beyond a simple "winner-takes-all" comparison. We argue that a comprehensive benchmark, including foundational and intermediate protocols, is crucial for several reasons: first, to quantify the real-world per-

doi:10.6342/NTU202503318

formance gaps predicted by asymptotic theory; second, to empirically validate the historical evolution of the field, demonstrating why modern designs are superior; and third, to uncover the nuanced engineering trade-offs and niche use cases that persist even for theoretically 'inferior' protocols. This work aims to provide a holistic, data-driven narrative that is valuable for both expert practitioners and newcomers to the field.

doi:10.6342/NTU202503318

Chapter 2 Background

Before delving into the complexity of lookup arguments, it is essential to establish an understanding of the underlying cryptographic concepts. This section introduces the core properties of Zero-Knowledge Proofs (ZKP) [14] and several key ideas that form the foundation of modern proof systems, such as Succinct Non-interactive ARgument of Knowledge (SNARK) [15, 16]. These fundamental concepts provide the necessary background for understanding more advanced techniques.

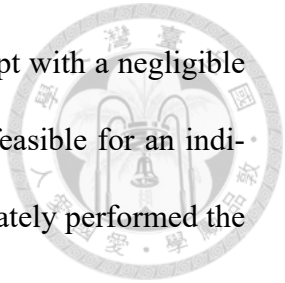
2.1 Key Properties of Zero-Knowledge Proofs

ZKPs are characterized by three essential properties:

Completeness: A ZKP protocol exhibits completeness if, given a true statement, an honest prover can successfully convince an honest verifier of its truth. In other words, when the prover genuinely possesses the knowledge or has accurately performed the computation, they should be able to generate a proof that will be accepted by the verifier.

Soundness: A ZKP protocol demonstrates soundness if, given a false statement, no dis-

doi:10.6342/NTU202503318



honest prover can convince an honest verifier of its truth, except with a negligible probability. This implies that it should be computationally infeasible for an individual who lacks the requisite knowledge or who has not accurately performed the computation to produce a proof that the verifier will accept.

Zero-Knowledge: A ZKP protocol possesses the property of zero-knowledge if, when the statement is true, the verifier acquires no information beyond the fact that the statement is indeed true. This is the fundamental privacy-preserving attribute of ZKPs. The proof does not disclose any additional information concerning the underlying knowledge or computation.

These core properties define what a ZKP system achieves. Beyond these foundational characteristics, ZKPs can also be categorized based on their interaction model.

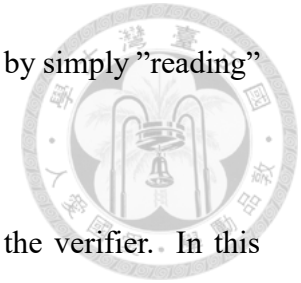
2.2 Interactive and Probabilistic Proofs: Incorporating Interaction and Randomness

In the realm of advanced proof systems, two fundamental concepts significantly depart from traditional, static notions of verification: interaction and randomness. These elements are central to the framework of Interactive and Probabilistic Proofs [14, 17].

The first key ingredient is Interaction. Unlike conventional proof verification where a verifier passively examines a provided proof, interactive proof systems involve a dynamic exchange. The verifier actively engages in a non-trivial dialogue or protocol with the prover. This interaction allows the verifier to query the prover and gain conviction about

doi:10.6342/NTU202503318

the truth of a statement through a structured conversation, rather than by simply ”reading” a pre-compiled document.



The second crucial component is Randomness on the part of the verifier. In this paradigm, the verifier is a randomized algorithm, often conceptualized as having the ability to perform actions akin to ”tossing coins” as a primitive operation during the verification process. This introduction of randomness means that the verifier's decision to accept or reject a proof is not necessarily deterministic. Consequently, there is a small, controlled probability that the verifier might err, either by rejecting a true statement or, less commonly in well-designed systems, accepting a false one. The power of such systems lies in the ability to make this error probability arbitrarily small.

These two elements—direct interaction between the prover and verifier, and the verifier's use of randomness—fundamentally redefine the verification process, leading to powerful and often more efficient proof systems for complex computational problems.

2.3 Arithmetic circuit

When discussing the transformation of computational problems into a format amenable to certain proof systems like SNARK, arithmetic circuits offer a fundamental and crucial model. An arithmetic circuit C is typically defined over a finite field $F = \{0, 1, \ldots, p-1\}$, where p is a prime number greater than 2. Such a circuit can be viewed as a function $C : F^n \to F$, which takes n inputs from the field F and produces a single output also within F.

Structurally, an arithmetic circuit is a directed acyclic graph (DAG). In this graph,

doi:10.6342/NTU202503318

internal nodes represent arithmetic operations, commonly addition (+), subtraction (-), or multiplication ($\times$) gates. The inputs to the circuit (or leaf nodes) are labeled with either constants (such as 1) or input variables $x_1, x_2, \ldots, x_n$.

A key characteristic of arithmetic circuits is that each circuit naturally defines an n-variate polynomial over the field F. The structure of the circuit itself provides an " evaluation recipe" for this polynomial. For instance, the circuit depicted in Figure 2.1 computes its output polynomial through a composition of intermediate addition, subtraction, and multiplication gates.

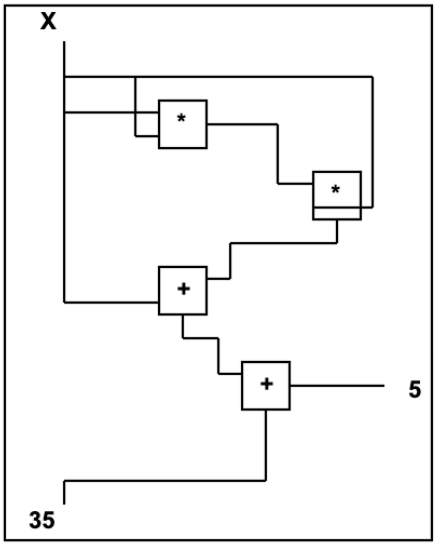


Figure 2.1: A diagram illustrating the arithmetic circuit.

The size of an arithmetic circuit, often denoted as $|C|$, is defined by the total number of gates it contains. This size is a common metric for the complexity of the computation represented by the circuit. In the context of zero-knowledge proofs, converting a computational problem into an arithmetic circuit (a process known as arithmetization) is a primary step in constructing a proof, and the circuit's size directly influences the proof generation time and resource consumption.[1]

[1] https://rdi.berkeley.edu/zk-learning/assets/Lecture2-2023.pdf

doi:10.6342/NTU202503318

2.4 What's NARK, SNARK, and zkSNARK

Non-interactive Arguments of Knowledge (NARKs), particularly those involving a preprocessing step, are foundational cryptographic systems. A NARK allows a prover to convince a verifier of the knowledge of a secret witness w that, along with a public statement x, satisfies a given public arithmetic circuit C. This relationship is often expressed as $C(x, w) = 0$, where x is a public statement in F^n (for some field F) and w is a secret witness in F^m.

The system typically involves a Preprocessing (or Setup) phase, denoted by an algorithm S. This algorithm takes the circuit C as input and generates public parameters, which can be split into prover parameters (pp) and verifier parameters (vp).

The core interaction then proceeds non-interactively:

- The Prover, using its parameters pp, the public statement x, and its secret witness w, executes a proving algorithm $P(pp, x, w)$ to produce a proof π.
- The Verifier, using its parameters vp, the public statement x, and the received proof π, executes a verification algorithm $V(vp, x, \pi)$ to either accept or reject the proof.

Formally, a preprocessing NARK can be defined as a triple of algorithms (S, P, V):

$S(C) \to (pp, vp)$: The setup algorithm generates public parameters for the prover and verifier based on the circuit C.

$P(pp, x, w) \to \pi$: The prover algorithm takes the prover parameters, public statement, and secret witness to generate a proof π.

doi:10.6342/NTU202503318

$V(vp, x, \pi) \rightarrow$ accept or reject: The verifier algorithm takes the verifier parameters, public statement, and proof to decide on its validity. It's often assumed in the security analysis of such systems that all algorithms and any adversary have access to a random oracle.

Building upon this, a SNARK (Succinct Non-interactive ARgument of Knowledge) [15] is a special type of preprocessing NARK that offers crucial efficiency properties. A SNARK is also defined by a triple (S, P, V), where S is the same setup algorithm. However, P and V have additional characteristics:

- The proving algorithm $P(pp, x, w)$ produces a short proof π. This "succinctness" means the length of the proof, len(π), is sublinear with respect to the size of the witness $|w|$. For example, if the witness has n elements, the proof size might be proportional to $\log n$ or $\sqrt{n}$.
- The verification algorithm $V(vp, x, \pi)$ is fast to verify. The time taken for verification, time(V), is typically sublinear in the size of the circuit $|C|$ and might depend on the size of the public statement $|x|$ (often denoted as $O_\lambda(|x|, \text{sublinear}(|C|))$, where λ is the security parameter). An example of a sublinear function is $f(n) = \sqrt{n}$.

These properties of succinct proof size and fast verification make SNARKs particularly attractive for applications where communication bandwidth and verifier computation are constrained, such as in blockchain systems.

Finally, a widely sought-after variant is the zk-SNARK. This refers to a SNARK that additionally incorporates the property of zero-knowledge. This means that the proof not only convinces the verifier of the statement's truth but does so without revealing any

doi:10.6342/NTU202503318

information about the witness (w) beyond the veracity of the statement itself.

In summary, these cryptographic arguments form a hierarchy of concepts, where SNARKs build upon NARKs by adding succinctness, and zk-SNARKs further enhance them with zero-knowledge. While the term "zk-SNARK" has become ubiquitous, it is crucial to recognize which of these properties is being leveraged in a given context. In applications focused on privacy, the zero-knowledge aspect is paramount. However, for blockchain scaling solutions like ZK-rollups—a central topic of this thesis—the driving force is the succinctness ('S') of the proof, which allows a Layer 1 chain to efficiently verify a large batch of off-chain transactions. Understanding this distinction is fundamental as we proceed to explore the other critical components that constitute these powerful proof systems.

2.5 The Preprocessing Setup (S) in SNARKs

The setup phase, denoted as $S(C)$, is a critical preliminary step in SNARKs, responsible for generating public parameters (pp, vp) required by the prover and verifier, respectively, for a given computation C. This setup often involves the use of random bits, denoted as r. The nature and handling of this randomness lead to different types of setup procedures, each with distinct trust assumptions and properties.

2.5.1 Types of Preprocessing Setups

The methodology for generating these parameters can be broadly categorized as follows:

doi:10.6342/NTU202503318

Trusted Setup per Circuit: In this model, the setup $S(C; r)$ utilizes random bits r that are specific to a particular circuit C. It is paramount that this randomness r be kept secret, especially from the prover. If the prover were to learn these secret random bits r, the security of the system could be compromised, potentially allowing the prover to generate convincing proofs for false statements. This necessitates a new trusted setup ceremony for each distinct circuit or program.

Trusted but Universal (Updatable) Setup: To overcome the limitation of per-circuit setups, universal or updatable setups have been developed. In this approach, a portion of the secret randomness r is independent of any specific circuit C. The setup process S can be seen as a two-stage procedure:

1. $S_{\text{init}}(\lambda; r) \rightarrow gp$: A one-time initial setup is performed using a security parameter λ and secret randomness r to generate global parameters (gp). This secret r must be kept secure.

2. $S_{\text{index}}(gp, C) \rightarrow (pp, vp)$: Subsequently, for any specific circuit C, these global parameters (gp) can be used by a deterministic algorithm S_{index} to derive the specific public parameters (pp, vp) for that circuit. This model allows for a single, initial trusted ceremony, after which parameters for multiple different circuits can be generated without new secret randomness. "Updatable" variants further allow multiple parties to contribute to the initial randomness in a way that as long as at least one party is honest and discards their randomness, the overall setup is secure.

Transparent Setup: Considered an ideal scenario in terms of trust assumptions, a transparent setup is one where the generation of public parameters $S(C)$ does not rely

doi:10.6342/NTU202503318

on any secret data or randomness that needs to be kept hidden and later destroyed. In such systems, all data used for the setup is publicly available, or the randomness is generated in a publicly verifiable way (e.g., derived from a public beacon or using nothing-up-my-sleeve numbers). This entirely obviates the need for a trusted setup ceremony, eliminating concerns about the potential compromise or mishandling of secret setup parameters. Consequently, transparent setups are often preferred as they offer stronger and more verifiable security guarantees regarding the setup phase.

The choice of setup mechanism has significant implications for the practicality, security, and trust model of a SNARK system. While trusted setups were common in earlier SNARK constructions, ongoing research increasingly focuses on developing and improving transparent setup methodologies.

2.6 General Construction Paradigm for SNARKs

The construction of SNARKs for general circuits often follows a two-step paradigm, combining distinct cryptographic and information-theoretic primitives. This approach can be visualized as taking two primary ingredients and ”compiling” or ”combining” them to yield the desired SNARK.

The two core components in this paradigm are:

1. A Functional Commitment Scheme: This is a cryptographic object. A functional commitment scheme allows a party to commit to a function (or a polynomial representing the computation) in such a way that they can later prove properties about

doi:10.6342/NTU202503318

this function (e.g., its evaluation at a certain point) without necessarily revealing the entire function. The commitment is binding (the committer cannot change the function after commitment) and often hiding (the commitment does not reveal the function).

2. A Compatible Interactive Oracle Proof (IOP): This is an information-theoretic object. An IOP is a type of interactive proof system where the prover sends messages that can be thought of as oracles (functions). The verifier, instead of reading these oracles entirely, makes a limited number of queries to them. The security of an IOP is information-theoretic, meaning it does not rely on computational hardness assumptions but rather on the properties of information and probability. For use in SNARK construction, this IOP needs to be ”compatible” with the chosen functional commitment scheme.

These two components are then brought together—conceptually, one might imagine them being processed or compiled (as suggested by the blender anology in some presentations) —to produce the SNARK for general circuits. The functional commitment scheme is used to compile the IOP into a concrete, non-interactive argument, making the prover’s messages (oracles) succinct and efficiently verifiable. The properties of the commitment scheme ensure the cryptographic soundness of the resulting SNARK, while the IOP provides the underlying proof structure and efficiency.

Further details on functional commitment schemes and Interactive Oracle Proofs would typically follow to elaborate on their specific properties and how their combination achieves the desired SNARK characteristics (succinctness, non-interactivity, and knowledge soundness).

doi:10.6342/NTU202503318

2.7 Functional Commitment Scheme

A core cryptographic primitive underpinning the construction of many Succinct Non-interactive Arguments of Knowledge (SNARKs) is the functional commitment scheme. To understand these, it's helpful to first consider basic data commitments. These typically involve a `commit(m, r)` algorithm that produces a commitment `com` from a message `m` and randomness `r` (i.e., `commit(m, r)` $\rightarrow$ `com`), and a `verify(m, com, r)` algorithm that subsequently checks this (i.e., `verify(m, com, r)` $\rightarrow$ `accept or reject`). Such schemes are characterized by being binding, meaning a committer cannot feasibly open a single commitment to two different messages, and hiding, where the commitment `com` reveals little information about the committed message `m`. A standard construction employs a cryptographic hash function H, where $com \coloneqq H(m, r)$, deriving its security from the properties of H (formally, $H : \mathcal{M} \times \mathcal{R} \rightarrow \mathcal{T}$).

Functional commitments elevate this concept by enabling commitment not just to static data, but to an entire function f chosen from a predefined family $\mathcal{F} = \{f : X \rightarrow Y\}$. In this paradigm, a prover selects $f \in \mathcal{F}$ and randomness r, sends a commitment $com_f \leftarrow \text{commit}(f, r)$ to a verifier. Subsequently, for any input $x \in X$, the prover can provide a claimed output $y \in Y$ along with a proof π. This proof π is crucial as it convinces the verifier that $f(x) = y$, that f indeed belongs to the family $\mathcal{F}$, and that this is the same function to which com_f corresponds.

Formally, a functional commitment scheme for a function family $\mathcal{F}$ is defined by three algorithms:

- `setup`$(1^\lambda) \rightarrow gp$: This algorithm takes a security parameter λ and outputs public

doi:10.6342/NTU202503318

parameters gp.

- `commit`$(gp, f, r) \to com_f$: Given the public parameters gp, a function $f \in \mathcal{F}$, and randomness r, this algorithm produces a commitment com_f. The scheme is binding and often, optionally, hiding.

- `eval(Prover P, Verifier V)`: This defines the interaction for proving and verifying an evaluation. The prover $P(gp, f, x, y, r)$ generates a short proof π for a claimed evaluation $f(x) = y$. The verifier $V(gp, com_f, x, y, \pi)$ then checks this proof and outputs `accept` or `reject`.

This `eval` step essentially functions as a proof system (often a zero-knowledge SNARK, or zk-SNARK) for the relation asserting that $f(x) = y$, $f \in \mathcal{F}$, and that com_f is the commitment to f using gp and some r.

The versatility of functional commitments is demonstrated by several important types crucial for modern cryptography:

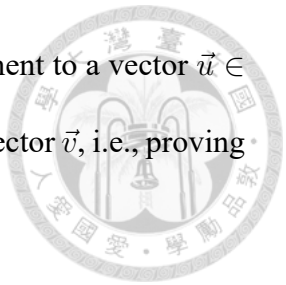
- Polynomial commitments [18, 19] allow for committing to univariate polynomials $f(X)$ over a field $\mathbb{F}_p$ of degree at most d (e.g., $f(X) \in \mathbb{F}_p[X]_{\leq d}$) and proving their evaluations at specified points.

- Multilinear commitments [19, 20] extend this to multilinear polynomials in k variables over $\mathbb{F}_p$ (e.g., $f(X_1, \ldots, X_k)$ where the degree in each variable is at most 1), allowing proofs of evaluation on specific input vectors.

- Vector commitments [21, 22] (e.g., Merkle trees) enable commitment to a vector $\vec{u} = (u_1, \ldots, u_d) \in \mathbb{F}_p^d$. They allow for proving individual elements, i.e., $f_{\vec{u}}(i) = u_i$ for a given index i.

doi:10.6342/NTU202503318

- Inner Product Commitments (IPAs) [23, 24] facilitate commitment to a vector $\vec{u} \in \mathbb{F}_p^d$ and allow the prover to open an inner product with a public vector $\vec{v}$, i.e., proving $f_{\vec{u}}(\vec{v}) = \langle \vec{u}, \vec{v} \rangle$.



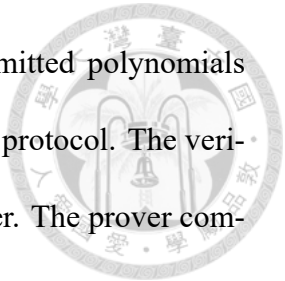
These varied schemes provide a rich toolkit for constructing advanced cryptographic protocols by enabling parties to verifiably bind themselves to complex mathematical objects and their properties.

2.8 Schwartz-Zippel Lemma and Fiat-Shamir Transform to Enable Polynomial Zero Test and Equality Test

A fundamental principle underpinning many cryptographic protocols, including Succinct Non-interactive Arguments of Knowledge (SNARKs) for polynomial properties, is the Schwartz-Zippel Lemma [25, 26]. This lemma states that for any non-zero polynomial $f \in \mathbb{F}_p[X]$ of degree at most d, the probability that $f(r) = 0$ for a randomly chosen point $r \leftarrow \mathbb{F}_p$ is at most d/p. When the field size p is significantly larger than d (e.g., $p \approx 2^{256}$ and $d \leq 2^{40}$), the ratio d/p becomes negligible. Consequently, if $f(r) = 0$ for a random r, f must be the identically zero polynomial with overwhelmingly high probability (w.h.p). This provides a simple probabilistic zero test.

This principle naturally extends to an equality test for two polynomials $f, g \in \mathbb{F}_p[X]$ of degree at most d: if $f(r) = g(r)$ for a random r, then $f \equiv g$ w.h.p., because this is equivalent to testing if the polynomial $(f - g)(X)$, also of degree at most d, is zero. The Schwartz-Zippel lemma also generalizes to multivariate polynomials, where d represents the total degree.

doi:10.6342/NTU202503318



An interactive protocol for verifying the equality of two committed polynomials (given commitments com_f, com_g) typically proceeds as a public coin protocol. The verifier selects a random challenge point $r \leftarrow \mathbb{F}_p$ and sends it to the prover. The prover computes $y \leftarrow f(r)$ and $y' \leftarrow g(r)$, then returns (y, π_f) and (y', π_g) to the verifier. Here, π_f and π_g are proofs, generated using the underlying polynomial commitment scheme, verifying the correctness of these evaluations with respect to com_f and com_g, respectively. The verifier accepts if both proofs π_f, π_g are valid and if $y = y'$.

To transform this interactive protocol into a non-interactive argument (a SNARK), the Fiat-Shamir transform [27] is applied. This heuristic employs a cryptographic hash function $H : \mathcal{M} \to \mathcal{R}$ (modeled as a random oracle, and often instantiated with functions like SHA256 in practice). Instead of receiving r from the verifier, the prover computes the challenge r autonomously by hashing a public message x (which could include com_f, com_g, and other relevant contextual information): $r \leftarrow H(x)$. The prover then calculates $y \leftarrow f(r)$, $y' \leftarrow g(r)$, generates the evaluation proofs π_f, π_g, and sends the tuple (y, y', π_f, π_g) to the verifier. The verifier also computes $r \leftarrow H(x)$ independently and performs the same verification checks. This non-interactive construction constitutes a SNARK for polynomial equality, provided that the ratio d/p is negligible and the hash function H behaves as a random oracle.

2.9 IOP, Polynomial IOP

The second pivotal component in the general paradigm for constructing SNARKs for arbitrary circuits, complementing functional commitment schemes, is the Functional Interactive Oracle Proof (F-IOP), or more generally, an Interactive Oracle Proof (IOP)

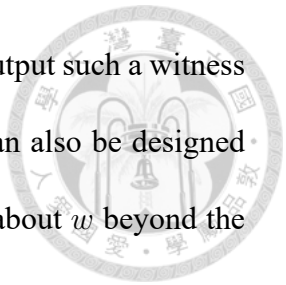
doi:10.6342/NTU202503318

[28, 29]. The primary objective of an F-IOP is to "boost" a given functional commitment scheme to achieve a SNARK capable of proving statements about general arithmetic circuits. For instance, a polynomial commitment scheme designed for polynomials in $\mathbb{F}_p[X]$ of degree at most d, when combined with a suitable Poly-IOP (an IOP tailored for polynomial properties), can yield a SNARK for any circuit C whose size or complexity (e.g., number of gates) is bounded by d. Formally, let $C(x, w)$ represent an arithmetic circuit where $x \in \mathbb{F}_p^n$ is the public input and w is the private witness; an F-IOP serves as a proof system to demonstrate the existence of such a witness w satisfying $C(x, w) = 0$.

The F-IOP typically begins with a Setup(C) phase that generates public parameters pp for the prover and vp for the verifier. The verifier's parameters vp might include oracles for certain initial functions (denoted in some contexts as $f_0, f_{-1}, \ldots, f_{-s}$) from the relevant function family $\mathcal{F}$. The interaction proceeds in rounds: the Prover $P(pp, x, w)$ sends an oracle representing a function $f_i \in \mathcal{F}$ to the Verifier $V(vp, x)$, who then responds by sampling a random challenge $r_i \leftarrow \mathbb{F}_p$ and sending it back to the prover. This exchange may repeat for t rounds. After all oracle messages $f_1, \ldots, f_t$ have been exchanged, the verifier performs a final decision step, often denoted as $\text{verify}^{f_0,\ldots,f_{-s},f_1,\ldots,f_t}(x, r_1, \ldots, r_{t'})$, which depends on the initial oracles (if any), the prover's oracles, the public input x, and all collected random challenges. A key feature is that the verifier can efficiently query any of the prover's oracles f_i at any desired point.

F-IOPs are characterized by several crucial properties. Completeness ensures that if a valid witness w exists such that $C(x, w) = 0$, the verifier will accept the proof with probability 1. (Unconditional) Knowledge Soundness, a defining property of IOPs, guarantees that if a (possibly malicious) prover convinces the verifier, then a corresponding witness w must exist and can be extracted. Specifically, an extractor algorithm, given ora-

doi:10.6342/NTU202503318



cle access to the prover's messages (e.g., $x, f_1, r_1, \ldots, r_{t-1}, f_t$), can output such a witness w. Optionally, if the goal is to construct a zk-SNARK, the F-IOP can also be designed to possess zero-knowledge, ensuring that the verifier learns nothing about w beyond the truth of the statement $C(x, w) = 0$.

To illustrate these concepts, consider a (somewhat contrived) example of a Poly-IOP designed to prove the relation $X \subseteq W$ for sets $X, W \subseteq \mathbb{F}_p$, framed as $C(X, W) = 0$. The prover, knowing X (public) and W (private witness), defines polynomials: $g(Z) := \prod_{x' \in X}(Z - x')$ (the vanishing polynomial for X, known to the verifier), $f(Z) := \prod_{w' \in W}(Z - w')$ (vanishing polynomial for W), and $q(Z) := f(Z)/g(Z)$. The assertion $X \subseteq W$ implies that $g(Z)$ must be a factor of $f(Z)$, meaning $q(Z)$ is a polynomial. The prover sends oracles for $f(Z)$ and $q(Z)$ (both asserted to be polynomials of degree at most d, where $d \geq |W|$). The verifier then picks a random point $r \leftarrow \mathbb{F}_p$, queries the oracles to obtain $f(r)$ and $q(r)$, computes $g(r)$ itself, and accepts if $f(r) = g(r) \cdot q(r)$. By the Schwartz-Zippel Lemma, this equality at a random point implies $f(Z) = g(Z)q(Z)$ as polynomials w.h.p., thus confirming $X \subseteq W$. The knowledge soundness is demonstrated by an extractor that, if the verifier accepts, can recover W by finding all the roots of the polynomial $f(Z)$ (obtained from its oracle). This F-IOP, when compiled with a polynomial commitment scheme, would allow the prover to commit to f and q, and then prove the evaluation $f(r) = g(r)q(r)$ non-interactively or with minimal interaction.

doi:10.6342/NTU202503318

2.10 Application of SNARK: Rollups as a Layer 2 Solution

2.10.1 The Need for Scalability and the Rise of Rollups

Many public blockchains face significant scalability challenges. For instance, Ethereum has a block time of around 12 seconds, while VISA can process over 6,000 transactions per second. There is a clear demand for blockchain systems that offer greater scalability. In this context, rollups [30, 31] have emerged as a notable solution aimed at enhancing blockchain scalability. These protocols aim to augment transaction throughput and diminish operational costs by offloading transaction execution from the primary blockchain layer (Layer 1), while concurrently leveraging the security mechanisms of the main chain for data availability and transaction settlement. Within the domain of rollups, two principal paradigms exist for delegating computational tasks: optimistic rollups and zero-knowledge (ZK) rollups.

2.10.2 Zero-Knowledge Rollups and a ZK-EVM/ZK-VM

Specifically, ZK rollups function as a Layer 2 scaling methodology that aggregates multiple transactions into a batch and subsequently submits a succinct zero-knowledge proof (ZKP) to the Layer 1 chain, thereby cryptographically verifying the validity of the aggregated transactions. This topic is unrelated to the aspects of privacy that zero-knowledge proofs (ZKP) effectively leverage but the succinctness properties that ZKP possess. This strategy substantially reduces the volume of transaction data that must be processed and verified by the main chain, resulting in improved transaction processing

doi:10.6342/NTU202503318

speeds and reduced transaction fees.

To facilitate the off-chain computation paradigm, it is necessary to deploy an off-chain prover that generates ZKPs and an on-chain verifier that validates these proofs. However, it is pertinent to note that general ZK rollups typically cannot directly provide proofs for arbitrary programs written in high-level blockchain domain-specific languages (DSLs) such as Solidity. Instead, they are configured to give proofs for custom, pre-defined logic that is specific to the rollup's design.

The process of generating a ZKP involves several critical steps. Initially, the computational statement to be proven, like the correct execution of a program, is transformed into a structured format that a zero-knowledge proof system can understand. This often involves representing the computation as an arithmetic circuit. An arithmetic circuit essentially breaks down the computation into a series of basic operations, typically addition and multiplication, forming a network of 'gates.' Different zero-knowledge proof protocols exist, each with its own specific way of handling these arithmetic circuits. These protocols define the rules and procedures for how the prover creates the proof and how the verifier checks it. Each protocol imposes its own constraints and syntax on how the circuit must be formulated. For example, one prominent protocol is PLONK. In the context of the PLONK protocol, the arithmetic circuit needs to be structured precisely according to PLONK's specifications, as PLONK primarily operates on addition and multiplication gates to perform its verification process. A corresponding interactive or non-interactive prover and verifier protocol, often classified as a zkSNARK (Succinct Non-interactive Argument of Knowledge), then governs the proof generation and verification procedures.

It bears repeating that general ZK rollups typically do not provide proofs for arbitrary

doi:10.6342/NTU202503318

smart contracts written in blockchain DSLs like Solidity. Consequently, concerted efforts have been dedicated to the development of Zero-Knowledge Ethereum Virtual Machines (ZK-EVMs) [32–34]. The overarching objective of a ZK-EVM is to construct arithmetic circuits that cryptographically attest to the correct execution of the Ethereum Virtual Machine, as illustrated in Figure 2.2. For each step of the EVM execution, the prover is tasked with demonstrating the relationship between the current execution state, the prior execution state, and the subsequent execution state. Subsequently, the verifier must validate this entire bundle of state transition attestations. Proving EVM execution requires proving the validity of individual EVM opcodes, such as `add`, `mul`, and `sub`. It should be noted that while basic arithmetic operations such as addition and multiplication map relatively straightforwardly onto arithmetic circuits (which frequently employ addition and multiplication gates), other opcode operations can introduce significant computational overhead for proof generation. For instance, operations such as bit shifts or manipulations of 32-bit integers (which often require decomposing the integer into its constituent bits, thereby necessitating approximately 32 constraints, one per bit, to ensure consistency with the binary representation) become computationally intensive.

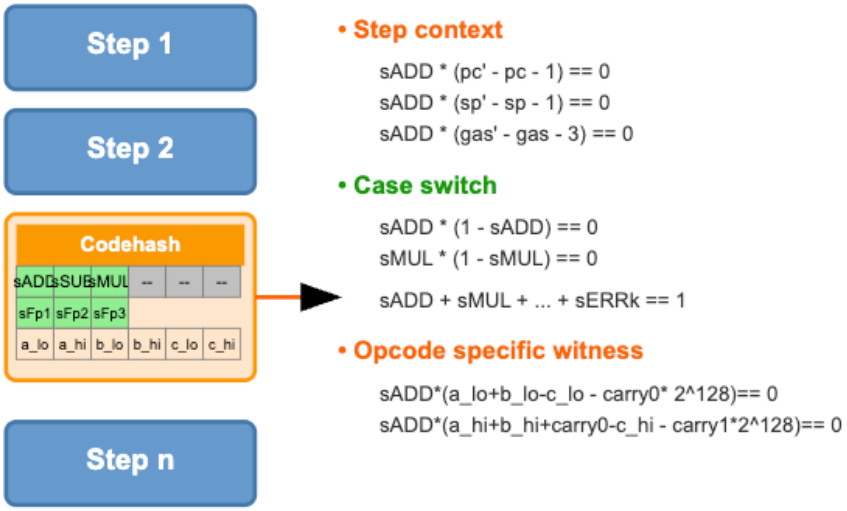


Figure 2.2: A diagram illustrating the ZK-EVM architecture. Source: [7]

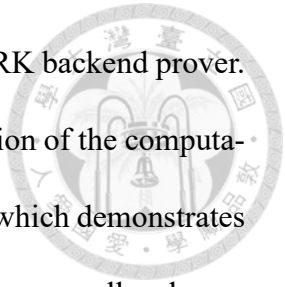
doi:10.6342/NTU202503318

In light of the inherent complexities associated with ZK-EVM design, a notable shift in research focus has transpired within the community, transitioning away from ZK-EVMs and toward Zero-Knowledge Virtual Machines (ZK-VMs) [35–37]. This alternative approach entails proving the execution of a compiled representation of an Ethereum node, such as Go Ethereum or Rust Ethereum, after it has been translated into a simpler instruction set architecture, such as RISC-V. This paradigm shift is largely motivated by the inherent simplicity and widespread adoption of the RISC-V instruction set. However, even with the simplification offered by ZK-VMs, proving every single instruction or operation in an execution trace remains a significant computational burden. This has led to the development of more specialized techniques, such as lookup arguments.

2.10.3 General Toolchain for SNARK Development

In practical applications, generating a SNARK proof for a specific computation typically follows a structured toolchain. The process often begins with developers writing a program using a domain-specific language (DSL) tailored for SNARK development. Examples of such DSLs include Circom [38], ZoKrates [39], Leo [40], Zinc [41], Cairo [42], and Noir [43], among others. These languages are designed to simplify the expression of computations in a way that is amenable to SNARK systems, abstracting away some of the complexities of direct circuit or constraint system construction. This high-level DSL program is then processed by a compiler. The compiler's role is to translate the program into a SNARK-friendly format, which is an intermediate representation that the SNARK proof system can directly operate on. Common examples of these formats include arithmetic circuits, Rank-1 Constraint Systems (R1CS), or even specialized bytecode like EVM (Ethereum Virtual Machine) bytecode if targeting blockchain applications.

doi:10.6342/NTU202503318



Once the computation is in this suitable format, it is fed into the SNARK backend prover. The prover, often depicted as a complex engine, takes this representation of the computation, along with the specific public input x and the private witness w (which demonstrates the correctness of the computation for that input), and performs what is generally a heavy computation. The result of this intensive proving process is a succinct proof, denoted as π. This proof π can then be delivered to a verifier to efficiently check the validity of the original computation without needing access to the private witness w. This pipeline facilitates the development of SNARK-based applications by providing higher-level tools and abstracting the intricate details of the underlying cryptographic primitives.

2.11 Introduction to Lookup Arguments

Traditionally, computational logic has been expressed through arithmetic circuits. Although more complex components, often termed ”gadgets,” can be constructed by combining these elementary gates to facilitate reusability, these gadgets are invariably expanded into their constituent addition and multiplication gates during circuit processing. This naturally prompts an inquiry into the feasibility of incorporating novel computational gates beyond simple addition and multiplication.

Significant advancements, particularly from research related to the Plonk proof system [44], have introduced the capability to define more sophisticated fundamental computational units. If the relationship between the inputs and outputs of a specific computation can be described by a predefined polynomial, this computation can be encapsulated as a basic unit. This innovation is known as a ”custom gate,” which can effectively be understood as a versatile, multi-input ”polynomial gate.”

doi:10.6342/NTU202503318

The evolution of gate design progressed further with the introduction of "lookup gates," notably described in the GW20 ([1]) paper. Unlike custom gates, the input-output behavior of lookup gates is not confined to polynomial relationships; instead, they can represent arbitrary, predefined relations. The conceptual basis for lookup gates is the use of a pre-established table, external to the circuit, where each row explicitly defines a valid input-output tuple for the intended operation. For instance, a table might enumerate specific valid tuples like $(in_1, in_2, in_3, out_1)$ without necessarily adhering to a simple algebraic formula. Given such a table, a lookup gate can be integrated into a circuit, and its operation is constrained to match one of the input-output entries present in this table, as shown in Figure 2.3. Such a mechanism is also commonly referred to as a lookup argument or lookup constraint.

When lookup gates are incorporated into a circuit within a proof system like Plonk, the protocol undertakes the verification of the gate's operational validity. This process involves consulting the predefined lookup table to ascertain if the observed input-output tuple from the gate's execution corresponds to an existing row. The gate's operation is deemed legitimate if a matching entry is found; otherwise, it is considered invalid, and the proof would be rejected.

In practical implementations, lookup gates find significant utility in representing bitwise operations efficiently. For example, an 8-bit XOR operation (mapping two 8-bit inputs to an 8-bit output) can be fully specified using a lookup table containing 2^{16} entries. Moreover, for cryptographic algorithms that extensively use bitwise operations, such as SHA256, the application of lookup arguments, sometimes facilitated by specialized structures like "spread tables" (which help decompose values for table lookups), can substantially improve the efficiency of representing these operations within a circuit. For

doi:10.6342/NTU202503318

instance, consider the bit shift operation mentioned earlier. Instead of creating a cumbersome arithmetic circuit to perform the bit shift, we can make a lookup table that lists all possible input values and their corresponding shifted outputs. Proving the correctness of a bit shift in a ZK-VM then reduces to showing that the input and output values from the execution are present in this table.

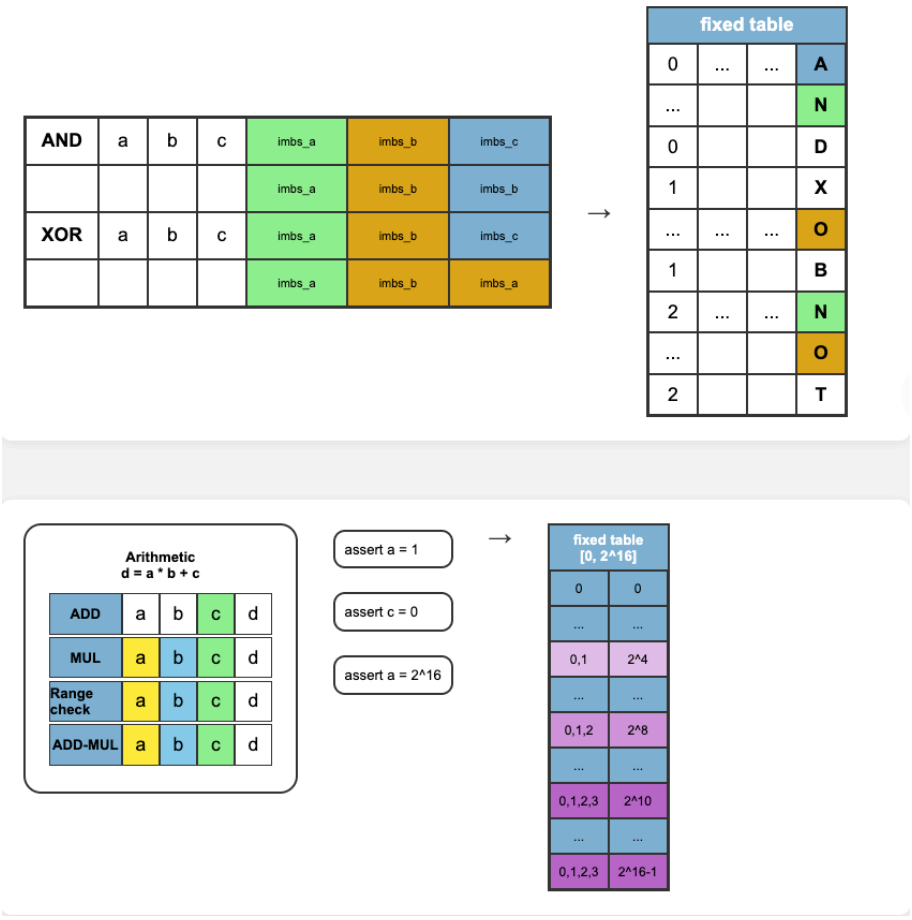


Figure 2.3: Illustration of a lookup argument using a precomputed table. Source: [8]

This technique of using lookup arguments allows for a dramatic simplification of the circuits needed for SNARK proofs, leading to significant performance improvements, especially for operations that are expensive to represent directly in arithmetic circuits.

doi:10.6342/NTU202503318

2.12 Lookup Argument Example

2.12.1 Range Proof

Two primary methodologies are employed for constructing range proofs in zero-knowledge circuits: a lookup-based approach and a bit decomposition approach.

2.12.1.1 Membership Testing via Lookup Argument

This method transforms the range proof problem into a membership test within a predefined set. The core idea is that if a value x is present in a lookup table containing all valid values for a given range, it is de facto within that range.

Proof Steps and Formulas A public lookup table, denoted as T, is pre-computed to contain all possible values for a specific bit-length. For an 8-bit integer, the table is defined as:

$$T = \{0, 1, 2, \ldots, 254, 255\}$$

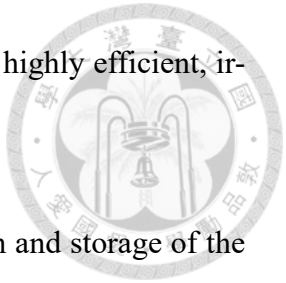
To prove that a value x is an 8-bit integer, the circuit imposes a single constraint:

$$x \in T$$

This constraint is a lookup argument, which can be efficiently verified by the underlying zero-knowledge proof protocol.

Characteristics

doi:10.6342/NTU202503318



- Advantages: The verification cost remains nearly constant and highly efficient, irrespective of the size of the range.
- Disadvantages: This approach necessitates the pre-computation and storage of the lookup table, which introduces additional setup costs and memory overhead.

2.12.1.2 Bit Decomposition

This alternative method involves decomposing the number into its constituent bits and applying algebraic constraints to validate its range. The central concept is that any n-bit number can be represented as a weighted sum of its n bits.

Proof Steps and Formulas For an 8-bit number x, it is decomposed into eight bits, $b_0, b_1, \ldots, b_7$. The proof consists of two main steps:

1. Prove that each b_i is a bit: The value of each bit must be either 0 or 1. This is enforced by the quadratic constraint:

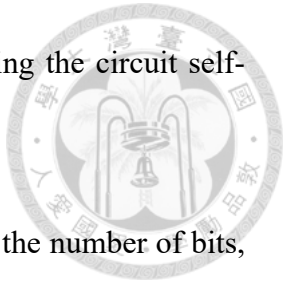
$$b_i \cdot (b_i - 1) = 0$$

2. Prove that the bit combination equals x: The weighted sum of the bits must reconstruct the original number x.

$$x = \sum_{i=0}^{7} b_i \cdot 2^i = b_7 \cdot 2^7 + b_6 \cdot 2^6 + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Characteristics

doi:10.6342/NTU202503318



- Advantages: No pre-computation or storage is required, making the circuit self-contained.
- Disadvantages: The number of constraints grows linearly with the number of bits, leading to an increased verification cost for larger ranges.

2.12.2 SHA-256

The implementation of the SHA-256 hash function [45] within a zero-knowledge circuit can be efficiently realized through the use of lookup arguments, primarily centered around a 16-bit lookup table. This design is optimized for larger circuits, requiring a minimum of 2^{16} rows and targeting a maximum constraint degree of 9.

A key component of this architecture is the `spread` table, which maps a 16-bit input to a 32-bit output where the original bits are interleaved with zeros. This table serves a dual purpose: it facilitates bitwise operations and implicitly performs range checks, thus obviating the need for a separate range check table.

doi:10.6342/NTU202503318

2.12.2.1 SHA-256 Compression Round

The SHA-256 algorithm performs 64 rounds of compression. Each round updates the 32-bit state variables A, B, C, D, E, F, G, H based on the following operations:

$$
\begin{aligned}
Ch(E,F,G) &= (E \wedge F) \oplus (\neg E \wedge G) \\
Maj(A,B,C) &= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \\
\Sigma_0(A) &= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \\
\Sigma_1(E) &= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25) \\
H' &= H + Ch(E,F,G) + \Sigma_1(E) + K_t + W_t \\
E_{new} &= \text{reduce}_6(H' + D) \\
A_{new} &= \text{reduce}_7(H' + \text{Maj}(A,B,C) + \Sigma_0(A))
\end{aligned}
$$

where $\ggg$ denotes a circular right shift and reduce_i handles a carry of up to i.

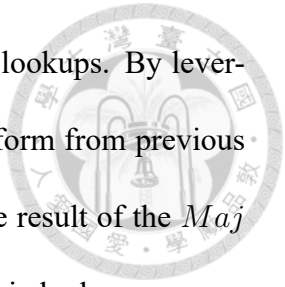
2.12.2.2 Core Functions Implementation via Lookup Arguments

Modular Addition To perform addition modulo 2^{32}, operands are decomposed into 16-bit chunks. For $a \boxplus b = c$, we have $(a_H, a_L) \boxplus (b_H, b_L) = (c_H, c_L)$, which is constrained using field addition:

$$\text{carry} \cdot 2^{32} + c_H \cdot 2^{16} + c_L = (a_H + b_H) \cdot 2^{16} + a_L + b_L$$

Each 16-bit chunk is range-checked by looking it up in the `spread` table's "dense" column. The carry value is constrained to its precise range using polynomial constraints.

doi:10.6342/NTU202503318



Maj Function The $Maj(A, B, C)$ function is implemented using 4 lookups. By leveraging the fact that the inputs A, B, C are available in their "spread" form from previous rounds, we compute their sum in the field: $M' = A' + B' + C'$. The result of the Maj function corresponds to the odd bits of this sum, which are extracted via lookups.

Ch Function The $Ch(E, F, G)$ function is implemented in 8 lookups. Similar to Maj, we assume the spread forms E', F', G' are available. The logic is implemented by computing two intermediate values, $P' = E' + F'$ and $Q' = (\texttt{spread}(2^{32} - 1) - E') + G'$. The sum of the odd bits of P' and Q' yields the Ch result.

Σ_0 and Σ_1 Functions The $\Sigma_0(A)$ and $\Sigma_1(E)$ functions are each implemented using 6 lookups. The 32-bit input is split into smaller bit-length pieces. The spread forms of these pieces are obtained via lookups in the `spread` table. The rotated and XORed result is then computed as a linear combination of these spread pieces, and the final output is extracted from the even bits of the resulting value, again using lookups.

Message Scheduling (σ_0 and σ_1) The message schedule expands the initial 16 words $(W_0, \ldots, W_{15})$ to 64 words. The expansion uses the σ_0 and σ_1 functions:

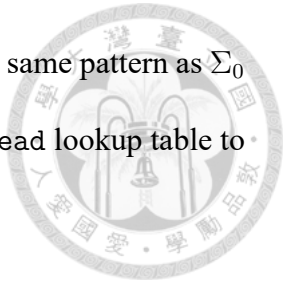
$$W_i = \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16}$$

$$\sigma_0(X) = (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3)$$

$$\sigma_1(X) = (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10)$$

doi:10.6342/NTU202503318

where $\gg$ is a right shift. The implementation of σ_0 and σ_1 follows the same pattern as Σ_0 and Σ_1, decomposing the input word into pieces and utilizing the `spread` lookup table to perform the bitwise operations efficiently.



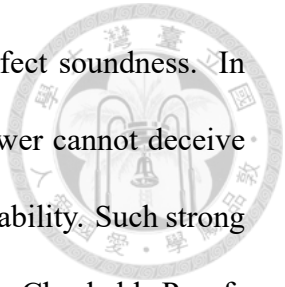
By systematically applying lookup arguments, primarily through the versatile `spread` table, the complex bitwise operations of SHA-256 are transformed into a set of efficient and verifiable circuit constraints. This demonstrates the power of lookup arguments in constructing complex cryptographic primitives within zero-knowledge proof systems.

2.13 Why Lookup "Argument" not Lookup "Proof"

The primary reason for this terminology lies in the difference between computational soundness and statistical/perfect soundness:

Argument: This term is used for proof systems where soundness relies on computational assumptions. These assumptions might include the difficulty of solving problems like discrete logarithms, the security of cryptographic pairings, or the collision resistance of hash functions. In such systems, a computationally bounded (typically polynomial-time) malicious prover has only a negligible probability of deceiving the verifier. However, an adversary with unlimited computational power could potentially break the soundness and create a false proof. Many practical systems, including SNARKs based on pairings (like those using KZG commitments, which are relevant to lookup protocols such as Plookup, Baloo, and CQ), fall into this category. They inherit the computational soundness from their underlying cryptographic primitives.

doi:10.6342/NTU202503318



Proof: This term is reserved for systems that offer statistical or perfect soundness. In these systems, even a prover with unlimited computational power cannot deceive the verifier, or can only do so with a statistically negligible probability. Such strong soundness is often achieved by systems based on Probabilistically Checkable Proofs (PCPs) or information-theoretic principles, like the Interactive Oracle Proofs (IOPs) underlying some FRI-based STARKs.

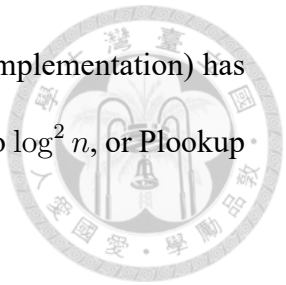
Since lookup protocols are typically built using polynomial commitment schemes (e.g., KZG, Pedersen, IPA) whose soundness is computational, the resulting lookup system inherits this computational soundness and is therefore classified as an ”argument.”

2.14 Motivation for Benchmarking Lookup Arguments

From a theoretical complexity perspective (Big O notation), we seem to have a good grasp of how different lookup protocols perform. So, why bother with time-consuming and effort-intensive benchmarking? Here are some key reasons:

1. Constant Factors and Lower-Order Terms Matter: Theoretical complexity (e.g., $O(n \log n)$, $O(n \log^2 n)$, $O(n^2)$) describes growth trends as the input size n approaches infinity, neglecting constant factors and lower-order terms. In practical applications, n may be large but not infinite. For real-world n values, a theoretically slower algorithm (like $O(n \log^2 n)$) with a very small constant factor might actually run faster than a theoretically superior algorithm (like $O(n \log n)$) with a large constant factor. Example: In our benchmark, Baloo is theoretically superior to Plookup for very large N. However, at $N = 8n$, its actual proof time is slower than

doi:10.6342/NTU202503318



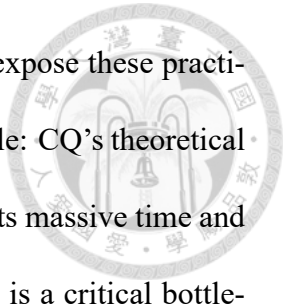
Plookup's. This strongly suggests that Baloo (in this specific implementation) has a very large constant factor or computational overhead related to $\log^2 n$, or Plookup has a very small constant factor.

2. Validating Theoretical Analysis: Theoretical analysis itself can contain errors, omissions, or oversimplified assumptions. Benchmarking serves as an experimental validation to check if theoretical predictions align with real-world scenarios. Example: Baloo's $O(n \log^2 n)$ theoretical complexity assumes the availability of efficient algorithms for arbitrary point set interpolation/evaluation. If these efficient algorithms are not present in the actual library or implementation, then the theoretical analysis doesn't apply to that specific implementation, and the benchmark reveals this discrepancy.

3. Assessing Implementation Quality and Library Impact: For the same algorithm, different implementations, programming languages, underlying libraries (e.g., for algebraic operations, FFT), and compiler optimizations can lead to vast performance differences. Benchmarking tests the performance of a specific implementation, not just an abstract algorithm. It helps uncover bottlenecks within particular libraries or code paths. Example: LogupGKR performed exceptionally well in this benchmark. This might be partly due to the inherent superiority of its protocol, but it could also reflect the highly efficient GKR/Sumcheck implementation (`crate::piop::gkr::fractional_sum_check`) it relies on.

4. Identifying Real-World Bottlenecks: Theoretical analysis often focuses on computationally intensive steps, but actual performance is also influenced by various factors such as memory access, cache efficiency, parallelism, data structure choices,

doi:10.6342/NTU202503318



and serialization/deserialization overhead. Benchmarking can expose these practical bottlenecks that theoretical analysis might overlook. Example: CQ's theoretical prover complexity is excellent, but the benchmark highlighted its massive time and potential space overhead during the preprocessing phase. This is a critical bottleneck that must be considered in actual deployment.

5. Comparing Different Trade-offs: Different lookup protocols involve various trade-offs across aspects like proving time, verification time, proof size, preprocessing time, memory usage, homomorphicity, and aggregatability. While theoretical analysis can compare a single metric (e.g., prover asymptotic complexity), benchmarking provides a more comprehensive, multi-dimensional performance picture. This helps developers make more informed choices based on specific application scenarios. For instance, if verification time is paramount, Plookup or LogupGKR might be chosen; if aggregatability is needed, CQ is the preferred option; and if extremely large fixed tables need to be processed and preprocessing is acceptable, CQ's fast prover might be very attractive.

6. Discovering Unexpected Behavior and Bugs: Sometimes, benchmarks reveal unexpected performance issues that were not anticipated theoretically, and they may even hint at bugs in the implementation. For example, if a protocol's performance grows far beyond its theoretical complexity with increasing input size, it could indicate a bug or a severe efficiency problem in the implementation.

doi:10.6342/NTU202503318

2.15 Rationale for Protocol Selection

The selection of protocols for this benchmark is guided by two main factors. Firstly, the field of lookup arguments is relatively new, with the chosen protocols representing the most significant proposals from the last few years. Secondly, each protocol introduces a distinct design philosophy, which makes a comparative analysis particularly insightful. Plookup [1] stands as the foundational work. Caulk [2] and Baloo [3] pioneered the concept of subtables to achieve sublinear prover complexity. CQ [4] explored the trade-offs of preprocessing and the power of the logarithmic derivative technique. LogupGKR [5] builds upon this by integrating the GKR protocol [46] for enhanced efficiency in a multilinear extension setting. Finally, Lasso [6] introduced a novel approach of decomposing large, structured tables into smaller, more manageable ones. This deliberate selection covers the key evolutionary steps and diverse design trade-offs in modern lookup arguments.

2.16 Theoretical Comparison of Lookup Arguments

The comparative evaluation of modern lookup argument protocols requires a multidimensional analysis, moving beyond singular metrics to capture the nuanced trade-offs inherent in their design. This section delineates the key characteristics, performance parameters, and mathematical notations used to systematically compare these protocols, providing a formal basis for their evaluation.

doi:10.6342/NTU202503318

Protocol Feature Dimensions

The protocols are assessed across several critical dimensions that reflect their computational cost, efficiency, and structural properties.

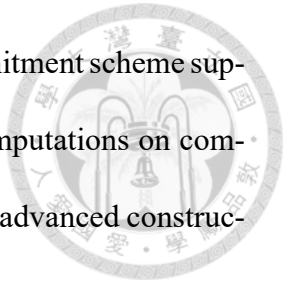
Prover Field & Group Complexity These metrics quantify the asymptotic complexity of operations required by the prover. Field Complexity refers to arithmetic operations (e.g., addition, multiplication) over the finite field, while Group Complexity pertains to more expensive elliptic curve operations (e.g., scalar multiplication) which are fundamental to the polynomial commitment scheme (PCS).

Sub-linearity This binary characteristic indicates whether the prover's computational workload scales sub-linearly with respect to the table size N. A protocol with this property ($i.e.$,, 'Yes') is critically advantageous for applications involving very large lookup tables, as its proving cost is not dominated by the table's size.

Pre-processing This refers to the one-time computational and storage costs incurred during a setup phase prior to any proof generation. Certain protocols require an expensive pre-processing step that is dependent on the entire table size N, whereas others, particularly those designed for structured tables ('struct.'), may require minimal to no pre-processing.

Proof Size This metric measures the size of the final proof object, typically in terms of the number of elliptic curve group elements ($\mathbb{G}_1$, $\mathbb{G}_2$) and finite field elements ($\mathbb{F}$). It is a crucial factor for applications constrained by on-chain storage or communication bandwidth.

doi:10.6342/NTU202503318



Homomorphic This indicates whether the protocol's underlying commitment scheme supports homomorphic properties. Homomorphism allows for computations on committed data, which is a powerful feature for efficiently handling advanced constructions such as multi-column (vector) lookups.

Aggregatable This property describes the protocol's capacity for efficiently aggregating multiple proofs into a single, compact proof that can be verified with a cost significantly lower than verifying each proof individually. This is essential for the scalability of systems like ZK-rollups.

Technique This identifies the core cryptographic or algebraic method upon which the protocol is built. Examples include permutation checks, sub-table extraction ('Ext.'), and logarithmic derivatives ('LogDeriv').

Commitment Scheme This specifies the type of Polynomial Commitment Scheme (PCS) required by the protocol. Some protocols are designed to be generic ('Any PCS'), while others depend on the specific properties of a particular scheme, such as Kate-Zaverucha-Goldberg (KZG).

Domain This describes the mathematical structure over which the polynomials in the protocol are defined, such as a univariate multiplicative subgroup ('Univar. Sub.') or a multilinear Boolean hypercube ('ML Hypercube').

Prover-defined $\mathbb{G}_2$ This indicates whether the generated proof contains a $\mathbb{G}_2$ group element that is dynamically computed by the prover. The presence of such an element ('Yes') can significantly complicate or inhibit proof aggregation, which often relies on fixed-base $\mathbb{G}_2$ points.

doi:10.6342/NTU202503318

Key Parameters and Notations

The following parameters and notations are used to articulate the complexity and size of the proofs.

N The size of the lookup table, representing the total number of entries.

m The number of lookup queries to be proven. Note that in the thesis text, n is often used interchangeably with m.

$\mathbb{G}_1, \mathbb{G}_2$ Elements of two distinct elliptic curve groups used in pairing-based cryptography.

$\mathbb{F}$ An element of the underlying finite field.

PCS, ML-PCS Acronyms for Polynomial Commitment Scheme and Multilinear Polynomial Commitment Scheme, respectively.

$O(\log N)\mathbb{F}$ This describes a component of the proof size consisting of a number of field elements that grows logarithmically with the table size N. This is characteristic of GKR-based protocols.

$O(\log m + \log N)\mathbb{F} + O(\log m)\mathbb{G}_1$ This describes a composite proof size. The total size includes a number of field elements growing logarithmically with both query count m and table size N, in addition to a number of $\mathbb{G}_1$ group elements growing logarithmically with the query count m. This structure is characteristic of the Lasso protocol.

doi:10.6342/NTU202503318

Table 2.1: Comparison of characteristics of various lookup protocols (horizontal full version)

Protocol	Features										
	Prover Field	Prover Group	Sub-linear?	Pre-process	Proof Size	Homo-morphic	Aggr-otable	Tech-nique	Commit-ment	Domain	Prover G_2?
Plookup	$O(N \log N)$	$O(N)$	No	None	5 G_1	No	Yes	Permutation	Any PCS	Univar. Sub.	No
Caulk	$O(m^2 + m \log N)$	$O(m^2)$	Yes	$O(N \log N)$	14 G_1, 1 G_2	Yes	No	Subtable Ext.	KZG	Arbitrary	Yes
Baloo	$O(m \log^2 m)$	$O(m)$	Yes	$O(N \log N)$	12 G_1, 1 G_2	Yes	No	Subtable+ Lin	KZG	Arbitrary	Yes
CQ	$O(m \log m)$	$O(m)$	Yes	$O(N \log N)$	8 G_1	Yes	Yes	LogDeriv+ CQ	KZG	Univar. Sub.	No
logUp-GKR	$O(m \log m)$	$O(M)$	No	None	cG_1/ $O(\log N) \times \mathbb{F}$	Yes	Yes	LogDeriv+ GKR	Any ML-PCS	ML Hypercube	No
Lasso	$O(m+N)$/ $O(cm)$	$O(m+N)$/ $O(cm)$	Yes	None (struct.)	$O(\log m + \log N) \times \mathbb{F}$ + $O(\log m)G_1$	Yes	Yes	Sparse Poly	Any ML-PCS	ML Hypercube	No

doi:10.6342/NTU202503318

2.17 Key Differences and Evolution of Lookup Arguments

2.17.1 Plookup ([1])

Plookup [1] is one of the earliest lookup protocols. It proves that a query vector $f \in \mathbb{F}^m$ is contained in a table vector $t \in \mathbb{F}^N$ (i.e., $\{f_j\} \subseteq \{t_i\}$).

Plookup's core idea is based on permutation checks over sorted vectors. To handle multisets, a common technique, particularly in later integrations like Plonkup, is to use a randomized difference check. Conceptually, given vectors t, f, an auxiliary sorted vector $s \in \mathbb{F}^{N+m}$ (which is $f \cup t$ sorted according to t), and bivariate polynomials derived from these vectors, one can establish the lookup claim. For instance, using random challenges β, γ:

$$F(\beta,\gamma) = (1+\beta)^m \prod_{j=1}^{m}(\gamma + f_j) \prod_{k=1}^{N-1}(\gamma(1+\beta) + t_k + \beta t_{k+1}) \tag{2.17.1}$$

$$G(\beta,\gamma) := \prod_{k=1}^{m+N-1}(\gamma(1+\beta) + s_k + \beta s_{k+1}) \tag{2.17.2}$$

Then the following holds:

$$F(\beta,\gamma) \equiv G(\beta,\gamma) \iff (\{f_j\} \subseteq \{t_i\} \text{ AND } s = (f,t)) \tag{2.17.3}$$

The equivalence relies on unique factorization of polynomials and matching factors corresponding to elements from f and transitions in t and s.

doi:10.6342/NTU202503318

2.17.1.1 Definitions

- m (query count) and N (table size) are independent, though for practical implementation in a single proof system, they are often padded to fit within a domain of size 2^k.
- $H = \{g, \ldots, g^N = 1\}$ is a multiplicative subgroup of order N in $\mathbb{F}$.
- For a vector $p \in \mathbb{F}^N$, $p(x) \in \mathbb{F}[X]_{<N}$ is its polynomial interpolation over H, so $p_i = p(g^i)$.
- $L_i(x) \in \mathbb{F}[X]_{<N}$ is the i-th Lagrange polynomial on H.
- $s \in \mathbb{F}^{2N-1}$ is (f, t) sorted by t. (Assuming $m = N - 1$ for this vector length).

Polynomial Representation: To process the vector s within the polynomial framework of zero-knowledge proofs, the protocol must convert this vector into polynomial form. Since the length of s (which is $n + d$) typically exceeds the size of the evaluation domain H (a multiplicative subgroup of size $n + 1$ used by the protocol), it is impossible to represent s completely using a single polynomial.

The solution is to partition the long vector s into two segments and represent them using two independent polynomials h_1 and h_2:

- $h_1(x)$: This polynomial, when evaluated at points in the multiplicative subgroup H (i.e., at g^i), yields values corresponding to the first half of the vector s. Specifically, $h_1(g^i) = s_i$ for $i = 1, \ldots, N$.
- $h_2(x)$: This polynomial, when evaluated at points in H, yields values corresponding to the second half of the vector s. Specifically, $h_2(g^i) = s_{N+i-1}$ for $i = 1, \ldots, N$.

doi:10.6342/NTU202503318

This polynomial decomposition enables the protocol to handle vectors that exceed the domain size while maintaining the algebraic structure necessary for efficient zero-knowledge verification.

2.17.1.2 The Protocol

1. Prover: Computes and commits to polynomials $h_1(x), h_2(x) \in \mathbb{F}[X]_{<N}$ such that for $i = 1, \ldots, N$:

$$h_1(g^i) = s_i \tag{2.17.4}$$

$$h_2(g^i) = s_{N+i-1} \tag{2.17.5}$$

2. Verifier: Sends random challenges $\beta, \gamma \in \mathbb{F}$ to the prover.

3. Prover: Computes and commits to an accumulator polynomial $Z(x) \in \mathbb{F}[X]_{<N}$ such that $Z(g) = 1$, $Z(g^N) = 1$, and for $i = 2, \ldots, N-1$:

$$Z(g^i) = (1+\beta)^{i-1} \frac{\prod_{l=1}^{i-1}(\gamma + f_l)(\gamma(1+\beta) + t_l + \beta t_{l+1})}{\prod_{l=1}^{i-1}(\gamma(1+\beta) + h_1(g^l) + \beta h_1(g^{l+1}))(\gamma(1+\beta) + h_2(g^l) + \beta h_2(g^{l+1}))} \tag{2.17.6}$$

doi:10.6342/NTU202503318

4. Verifier: Checks the following identities for all $x \in H$:

$$L_1(x)(Z(x) - 1) = 0 \tag{2.17.7}$$

$$L_N(x)(Z(x) - 1) = 0 \tag{2.17.8}$$

$$L_N(x)(h_1(x) - h_2(gx)) = 0 \tag{2.17.9}$$

$$(x - g^N)Z(x)(1+\beta)(\gamma + f(x))(\gamma(1+\beta) + t(x) + \beta t(gx)) =$$
$$(x - g^N)Z(gx)(\gamma(1+\beta) + h_1(x) + \beta h_1(gx))(\gamma(1+\beta) + h_2(x) + \beta h_2(gx)) \tag{2.17.10}$$

The polynomial $Z(x)$ aggregates the ratio $F(\beta, \gamma)/G(\beta, \gamma)$. The checks ensure $Z(g) = Z(g^N) = 1$ and the correct accumulation at each step.

2.17.1.3 Integration with the Plonk Protocol

The table vector t is predefined and can be committed to during a preprocessing phase. In Plonk, lookups are treated as special types of gates. The query vector f is typically derived from a combination (folding) of Plonk's witness columns (w_a, w_b, w_c). A selector polynomial, $q_K(X)$, distinguishes lookup gates.

Preprocessing phase: Commitments to Plonk selectors $[q_L(X)], \ldots, [q_C(X)]$; lookup selector $[q_K(X)]$; permutation polynomials $[\sigma_a(X)], \ldots$; and table columns $[t_1(X)], \ldots$.

Protocol Steps (Simplified):

1. Round 1 (Witness commitments): Prover commits $[w_a(X)], [w_b(X)], [w_c(X)]$.

2. Round 2 (Table Folding Challenge): Verifier sends η.

doi:10.6342/NTU202503318

3. Round 3 (Lookup Vector Commitments): Prover constructs $t(X)$ (folded table $\sum \eta^j t_j(X)$) and $f(X)$ (where $f(\omega^i)$ is $\sum \eta^j w_j(\omega^i)$ if $q_K(\omega^i) = 1$, else a default value). Prover computes s, splits it into polynomial representations (e.g., $h_1(X), h_2(X)$ as above), and commits to commitments for s.

4. Round 4 (Challenges): Verifier sends (β_1, γ_1) for Plonk permutation, and (β_2, γ_2) for Plookup.

5. Round 5 (Accumulator Commitments): Prover commits Plonk accumulator $[z_{perm}(X)]$ and Plookup accumulator $[z_{lookup}(X)]$.

6. Round 6 (Quotient Challenge): Verifier sends α_{agg}.

7. Round 7 (Quotient Polynomial Commitment): Prover aggregates all constraints (Plonk arithmetic, permutation, Plookup) into a single polynomial, computes quotient $T_{quot}(X)$, commits $[T_{quot}(X)]$.

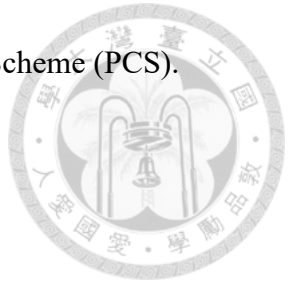
8. Subsequent Rounds (Opening Proof): Standard opening proof and verification.

2.17.1.4 Costs and Performance Characteristics

- Prover Asymptotics: $O(N \log N)$ field operations (for polynomial interpolation and FFTs) and $O(N)$ group operations.
- Sublinear in Table Size N?: No.
- Preprocessing: None, beyond standard PCS setup if table is not pre-committed.
- Proof Size (KZG): Approx. 5 $\mathbb{G}_1$ elements and 9 field elements.
- Verifier Work (KZG): Approx. 2 pairing evaluations.

doi:10.6342/NTU202503318

- Commitment Scheme: Can use any Polynomial Commitment Scheme (PCS).
- Domain: Univariate, over a multiplicative subgroup.



2.17.1.5 Generalizations and Optimizations

Plookup can be generalized to multiple witness polynomials $f_1, \ldots, f_w$ and tables $t_1, \ldots, t_w$ by using a random linear combination challenge α'_{agg} to aggregate them into single $f = \sum(\alpha'_{agg})^\ell f_\ell$ and $t = \sum(\alpha'_{agg})^\ell t_\ell$.

If the table represents a range of consecutive integers (e.g., $t_{l+1} = t_l + 1$), the accumulation for $Z(g^i)$ can be simplified.

The Plonkup protocol [47] integrates Plookup with Plonk, allowing lookups as generalized Plonk gates.

2.17.2 Caulk ([2])

Core Idea Introduced the first lookup argument with a prover workload that is sublinear in the size of the table (N). Caulk's core innovation is to extract a small sub-table containing only the necessary values and prove its correctness against the full table commitment using precomputed KZG proofs. It then proves, in zero-knowledge, that the query values are contained within this smaller, hidden sub-table.

Key Bottleneck While sublinear in the table size N, the prover's work has a quadratic dependency on the number of lookups ($O(m^2)$). This complexity arises from constructing a polynomial that checks the relationship between the query values and the extracted sub-table, making it inefficient for a large number of lookups in a single

doi:10.6342/NTU202503318

proof.

The core idea is to avoid operating on the full table polynomial $C(X)$. Instead, the prover constructs a commitment to a sub-table polynomial $C_I(X)$ that interpolates the subset of $\mathbf{c}$ corresponding to the values in $\mathbf{a}$. It uses precomputation to efficiently prove that $C_I(X)$ is a valid sub-table of $C(X)$. The main challenge is then to prove that the polynomial for the query values, $\phi(X)$, is correctly related to the hidden sub-table $C_I(X)$. This is achieved by creating an auxiliary "coordinate" polynomial $u(X)$ that maps the query domain to the hidden locations in the table domain and using it to check for value equality. This is verified with three core polynomial equations, which are batched for efficiency:

$$C(X) - C_I(X) = z_I(X)H_1(X) \tag{2.17.11}$$

$$z_I(u(X)) = z_{V_m}(X)H_2(X) \tag{2.17.12}$$

$$C_I(u(X)) - \phi(X) = z_{V_m}(X)H_3(X) \tag{2.17.13}$$

Equation (2.17.11) proves that $C_I(X)$ is a sub-table of $C(X)$. Equation (2.17.12) proves that the coordinates in $u(X)$ are roots of the sub-table's vanishing polynomial $z_I(X)$. Equation (2.17.13) proves that evaluating the sub-table at these coordinates yields the query values from $\phi(X)$.

2.17.2.1 Definitions

- m is the number of lookup queries; N is the size of the lookup table.

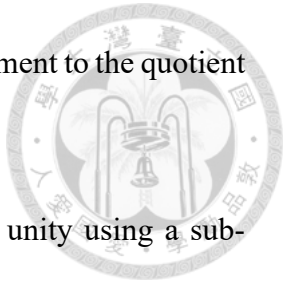
doi:10.6342/NTU202503318

- $H = \{1, \omega, \ldots, \omega^{N-1}\}$ is a multiplicative subgroup of order N used for the table domain.
- $V_m = \{1, \nu, \ldots, \nu^{m-1}\}$ is a multiplicative subgroup of order m used for the query domain.
- $\{\lambda_i(X)\}$ are the Lagrange basis polynomials over $H16$; $\{\mu_j(X)\}$ are the Lagrange basis polynomials over V_m.
- $C(X) = \sum_{i=1}^{N} c_i \lambda_i(X)$ is the polynomial interpolation of the table vector $\mathbf{c}$.
- $\phi(X) = \sum_{j=1}^{m} a_j \mu_j(X)$ is the polynomial interpolation of the query vector $\mathbf{a}$.
- $I \subset [N]$ is the set of secret indices in the table $\mathbf{c}$ that correspond to the values in $\mathbf{a}$.
- $C_I(X)$ is the polynomial interpolating the sub-vector $\mathbf{c}_I$ over the domain locations $\{\omega^{i-1}\}_{i \in I}$.
- $z_I(X) = \prod_{i \in I}(X - \omega^{i-1})$ is the vanishing polynomial for the sub-table locations.
- $u(X) = \sum_{j=1}^{m} \omega^{i_j - 1} \mu_j(X)$ is the coordinate polynomial that maps the j-th query to its location $\omega^{i_j - 1}$ in the table's domain.

2.17.2.2 The Protocol

1. Prover (Sub-table Extraction & Coordinate Mapping):
    - Identifies the subset of indices $I \subset [N]$ such that $\{c_i\}_{i \in I}$ contains all values from the query vector $\mathbf{a}$.
    - Constructs blinded, committed polynomials: $C_I(X)$ (sub-table values), $z_I(X)$ (sub-table vanishing poly), and $u(X)$ (coordinate mapping).

doi:10.6342/NTU202503318



- Using precomputed proofs, the prover computes a commitment to the quotient $H_1(X)$ for the sub-table check in Eq. (2.17.11).
- Proves that the outputs of $u(X)$ are valid N-th roots of unity using a sub-protocol (π'_{unity}).
- Sends commitments $[C_I]_1, [z_I]_1, [u]_1, [H_1]_2$ and the proof π'_{unity} to the verifier.

2. Verifier $\rightarrow$ Prover: Sends a random batching challenge χ.

3. Prover (Constraint Aggregation):

   - Aggregates the location check (Eq. (2.17.12)) and value check (Eq. (2.17.13)) into a single polynomial relation using χ: $z_I(u(X)) + \chi(C_I(u(X)) - \phi(X)) = z_{V_m}(X)H_2(X)$.
   - Commits to the combined quotient $[H_2]_1$ and sends it to the verifier.

4. Verifier $\rightarrow$ Prover: Sends a random evaluation point challenge α.

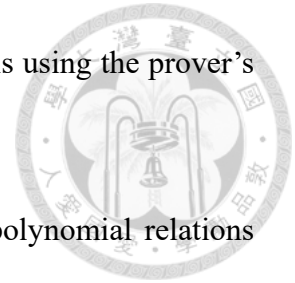
5. Prover $\rightarrow$ Verifier (Openings):

   - Computes evaluations $v_1 = u(\alpha)$ and $v_2 = z_I(v_1) + \chi C_I(v_1)$.
   - Creates and sends batched KZG opening proofs for $u(X)$ at α, for $z_I(X) + \chi C_I(X)$ at v_1, and for the aggregated relation at α.

6. Verifier (Verification):

   - Checks the sub-protocol proof π'_{unity}.
   - Checks the pairing equation for the sub-table extraction: $e([C]_1 - [C_I]_1, [1]_2) = e([z_I]_1, [H_1]_2)$.

doi:10.6342/NTU202503318



- Reconstructs commitments to the aggregated polynomials using the prover's commitments and challenges.
- Verifies the three KZG opening proofs to confirm all polynomial relations hold.

2.17.2.3 Costs and Performance Characteristics

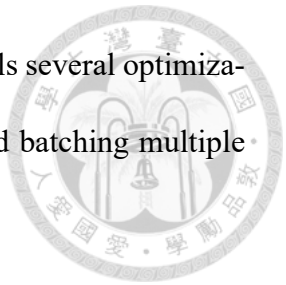
- Prover Asymptotics: $\tilde{O}(m^2 + m \log N)$. The m^2 term comes from composing polynomials of degree m.
- Sublinear in Table Size N?: Yes, which is its primary advantage over prior work.
- Preprocessing: Requires a one-time setup of $O(N \log N)$ to compute and store all single-opening KZG proofs for the table elements.
- Proof Size (KZG): Constant size. For the lookup argument, it is approximately $14\mathbb{G}_1$, $1\mathbb{G}_2$, and $4\mathbb{F}$ elements.
- Verifier Work (KZG): $O(\log(\log N))$ scalar operations and a constant number of pairings (4 pairings after batching).
- Commitment Scheme: Exclusively uses the KZG polynomial commitment scheme.
- Domain: Univariate polynomials over multiplicative subgroups of roots of unity.

2.17.2.4 Generalizations and Optimizations

Caulk supports lookups with repeated values, as the protocol proves subset membership without regard to order or multiplicity within the query vector. The protocol can also

doi:10.6342/NTU202503318

be used to generate proofs about ”sub-lookup tables”. The paper details several optimizations, such as batching multiple KZG openings at the same point and batching multiple pairing checks to reduce verifier work.



2.17.3 Baloo ([3])

The Evolution Baloo is the direct successor to Caulk, specifically designed to fix the $O(m^2)$ bottleneck. It achieved a nearly optimal prover time of $O(m \log^2 m)$.

Core Technique It retains Caulk's subtable extraction idea but replaces its inefficient lookup proof with a highly optimized framework based on linear relations and a Checkable Subspace Sampling (CSS) argument. This allows for proving the relationship between the lookup values and the extracted sub-table in quasi-linear time.

Remaining Weakness Its proof contains a prover-defined G2 element ($[z_I]_2$), which makes it difficult to aggregate multiple proofs recursively—a critical feature for scaling ZK systems.

Baloo [3] is an indexed lookup argument proving $\{a_i\} \subseteq \{t_i\}$ by reducing it to a matrix-vector relation $\mathbf{M}t_I = a$. Here t_I is a sub-vector of t, and $\mathbf{M}$ has unit-vector rows, ensuring a_i is a copy of $(t_I)_j$.

2.17.3.1 Core Components and Identities

Baloo protocol is constructed to prove a set of core algebraic relations between its constituent polynomials. The cryptographic checks using KZG commitments serve to verify these underlying identities.

doi:10.6342/NTU202503318

1. Subtable Equations: To prove $t_I \subseteq t$, the Prover constructs two quotient polynomials, $Q_I(X)$ and $W_{H\setminus I}(X)$, which satisfy the following identities:

$$t(X) - t_I(X) = Q_I(X) \cdot z_I(X) \tag{2.17.14}$$

$$z_H(X) = W_{H\setminus I}(X) \cdot z_I(X) \tag{2.17.15}$$

2. Matrix Structure and Consistency Equations: To prove the unit-vector structure of the matrix $\mathbf{M}$, the Prover constructs a quotient polynomial $q_2(X)$ satisfying the first identity below. The second identity is a direct evaluation check to ensure consistency between the row and column samplings.

$$e(X)(\beta - v(X)) + \frac{z_I(\beta)}{z_I(0)} v(X) = q_2(X) \cdot z_V(X) \tag{2.17.16}$$

$$d(\beta) = e(\alpha) \tag{2.17.17}$$

3. Dot Product Equation: To execute the sum-check argument for the dot product, the Prover constructs a quotient polynomial $q_1(X)$ that satisfies the following equation:

$$d(X)t_I(X) - a(\alpha) - Xg(X) = q_1(X) \cdot z_I(X) \tag{2.17.18}$$

2.17.3.2 The Protocol

1. P (Setup & Subtable): Commits $C_{t_I}, [z_I(X)]_1, [z_I(X)]_2, C_v$. Sends proofs for subtable extraction.

2. V $\rightarrow$ P: Sends challenge α.

doi:10.6342/NTU202503318

3. P (Row Sampling & Dot Product): Computes $d(X) = M(\alpha, X)$, commits C_d. Computes $g(X), q_1(X)$ for Eq. (2.17.18), commits C_g, C_{q_1}.

4. V $\rightarrow$ P: Sends challenge β.

5. P (Column Sampling & CP-Expansion): Computes $e(X) = M(X, \beta)$, commits C_e. Computes $q_2(X)$ for Eq. (2.17.16), commits C_{q_2}.

6. V $\rightarrow$ P: Sends challenge ζ.

7. P $\rightarrow$ V (Evaluations): Sends $a_\alpha, e_\alpha, d_\beta, z_{I0}, z_{I\beta}, e_\zeta$.

8. V $\rightarrow$ P: Sends batching challenges γ_i.

9. P $\rightarrow$ V (Openings): Sends batched KZG openings for identities from Eqs. (2.17.18), (2.17.16).

10. V (Verification): Verifies Subtable pairings, consistency $e_\alpha = d_\beta$ (Eq. (2.17.17)), and all KZG openings.

2.17.3.3 Costs and Performance Characteristics

- Prover Asymptotics: $O(n \log^2 n)$ field operations, $O(n)$ group operations (where n is number of queries). Uses non-subgroup operations.
- Sublinear in Table Size N?: Yes, with preprocessing.
- Preprocessing: $O(N \log N)$ for $\mathbb{G}_1$ elements and field operations (for precomputing all single-opening KZG proofs for the table).
- Proof Size (KZG): Moderate, e.g., 12 $\mathbb{G}_1$ elements, 1 $\mathbb{G}_2$ element.

doi:10.6342/NTU202503318

- Verifier Work (KZG): Moderate, e.g., 5 pairing evaluations.
- Homomorphic Table Commitment?: Yes, crucial for multi-column lookups via random linear combination.
- Aggregatable?: No (not easily, due to prover-defined $\mathbb{G}_2$ element $[z_I(X)]_2$).
- Commitment Scheme: Relies on KZG pairing-based commitments.
- Domain: Operates over arbitrary subsets of a subgroup for t_I.

Benchmark results indicate Baloo's prover performance can be slower than Plookup for certain N/n ratios or specific implementations, possibly due to high constant factors in field operations on arbitrary sets.

2.17.3.4 Generalizations and Variants

Baloo supports multi-list queries by preserving additively homomorphic properties of table commitments.

2.17.4 CQ (Cached Quotients) ([4])

Cached Quotients (CQ) [4] is a lookup argument using logarithmic derivatives to prove $\{f_j\} \subseteq \{t_k\}$. It avoids subtable extraction, operating on the full table t.

2.17.4.1 Core Idea and Key Equations

The logarithmic derivative of $P(X) = \prod(X - r_k)$ is $\sum \frac{1}{X - r_k}$. The lookup proof involves verifying at a random β:

doi:10.6342/NTU202503318

$$\sum_{j=1}^{n} \frac{1}{\beta - f_j} = \sum_{k=1}^{N} \frac{m_k}{\beta - t_k} \tag{2.17.19}$$

where m_k is multiplicity of t_k in f. This is done by checking three conditions using polynomials $A(X)$ (for $m_k/(\beta - t_k)$), $M(X)$ (for m_k), $T(X)$ (for t_k), $B(X)$ (for $1/(\beta - f_j)$), and $F(X)$ (for f_j):

(a) $A(X)(\beta - T(X)) - M(X) = Q_A(X)Z_{H_t}(X)$

(b) $B(X)(\beta - F(X)) - 1 = Q_B(X)Z_{H_f}(X)$

(c) $|H_t| \cdot A(0) = |H_f| \cdot B(0)$ (derived from $\sum A(\omega) = \sum B(\nu)$)

These are verified using KZG pairing checks for ((a))-((b)) and openings for ((c)).

2.17.4.2 The Protocol

1. V $\rightarrow$ P: Sends random challenge β.
2. P: Computes values $m_k/(\beta - t_k)$, $1/(\beta - f_j)$. Constructs polynomials $A(X), M(X), B(X), F(X), Q_A(X), Q_B(X)$.
3. P $\rightarrow$ V: Sends commitments $C_A, C_M, C_B, C_F, C_{Q_A}, C_{Q_B}$. ($C_T, C_{Z_{H_t}}, C_{Z_{H_f}}$ may be preprocessed/derived).
4. V $\rightarrow$ P: Requests openings (e.g., $A(0), B(0)$, and points for batched KZG).
5. P $\rightarrow$ V: Sends opened values and proofs.
6. V: Verifies KZG pairing identities for ((a)), ((b)), sum check ((c)), and all openings.

doi:10.6342/NTU202503318

2.17.4.3 Costs and Performance Characteristics

- Prover Asymptotics: $O(n \log n)$ field operations (FFT-based), $O(n)$ group operations.
- Sublinear in Table Size N?: Yes, with preprocessing.
- Preprocessing: $O(N \log N)$ for $\mathbb{G}_1$ elements and field operations (for "cached quotients"). Can be very large ("terabytes" for large N).
- Proof Size (KZG): Small, e.g., 8 $\mathbb{G}_1$ elements, 0 prover-defined $\mathbb{G}_2$ elements.
- Verifier Work (KZG): Moderate, e.g., 5 pairing evaluations.
- Homomorphic Table Commitment?: Yes.
- Aggregatable?: Yes, due to fixed $\mathbb{G}_2$ points.
- Commitment Scheme: Relies on KZG pairing-based commitments.
- Domain: Univariate, over multiplicative subgroups.

CQ offers strong asymptotic prover efficiency but has a very significant preprocessing cost.

2.17.4.4 Generalizations and Variants

The core technique focuses on efficient full-table operations for lookups.

doi:10.6342/NTU202503318

2.17.5 LogupGKR ([5])

LogupGKR [5] enhances the LogUp argument [48] by using the Goldwasser-Kalai-Rothblum (GKR) protocol [46] to prove LogUp's fractional sumchecks. This reduces prover commitment overhead, requiring commitment only to a multiplicity column. Assumes familiarity with MLEs, sumcheck, and GKR.

2.17.5.1 Core Argument and GKR Application

LogUp, like CQ, uses logarithmic derivatives. For witness MLEs $w_i(\vec{X})$, table $t(\vec{X})$, multiplicities $m(\vec{X})$ (all over $\mathbb{H}^n = \{\pm 1\}^n$), LogUp proves at random α:

$$\sum_{i=1}^{M} \left(\sum_{\vec{x} \in \mathbb{H}^n} \frac{1}{\alpha - w_i(\vec{x})} \right) - \sum_{\vec{x} \in \mathbb{H}^n} \frac{m(\vec{x})}{\alpha - t(\vec{x})} = 0 \tag{2.17.20}$$

This is a claim $\sum_{\vec{z}} \frac{p_{eff}(\vec{z})}{q_{eff}(\vec{z})} = 0$. GKR verifies this using a layered arithmetic circuit computing the sum via projective coordinates (a, b) for a/b.

Layer k computes $(p_k(\vec{x}), q_k(\vec{x}))$ from children in layer $k+1$:

$$p_k(\vec{x}) = p_{k+1}(\vec{x}, 1) q_{k+1}(\vec{x}, -1) + p_{k+1}(\vec{x}, -1) q_{k+1}(\vec{x}, 1) \tag{2.17.21}$$

$$q_k(\vec{x}) = q_{k+1}(\vec{x}, 1) q_{k+1}(\vec{x}, -1) \tag{2.17.22}$$

The GKR protocol interactively reduces claims from layer 0 down to layer n (inputs p_{eff}, q_{eff}). For the LogUp sum, p_{eff} and q_{eff} are constructed over $n + k'$ variables ($k' =$

doi:10.6342/NTU202503318

$\lceil \log_2(M+1) \rceil$) using Lagrange polynomials to select appropriate terms from Eq. (2.17.20).

2.17.5.2 The Protocol (GKR Interaction Summary)

1. Initial Claim: Prover claims $p_0 = 0, q_0 \neq 0$ for the sum (2.17.20).

2. Layer Reduction (Iterative): For each layer $k = 0, \ldots, n + k' - 1$:

   - Verifier provides random challenge λ_k. Prover and Verifier run sumcheck on $p_k(\vec{r}_k) + \lambda_k q_k(\vec{r}_k)$ (where $\vec{r}_k$ is point from previous round).
   - This sumcheck reduces the claim to evaluations at the child points (e.g., $p_{k+1}(\vec{\rho}_k, \pm 1),\ q_{k+1}(\vec{\rho}_k, \pm$
   - Verifier sends μ_k to combine these into a single point claim for $p_{k+1}(\vec{r}_{k+1}), q_{k+1}(\vec{r}_{k+1})$.

3. Final Step: Claims reduce to evaluations of base MLEs $t(\vec{X}), m(\vec{X}), w_i(\vec{X})$ at a random point. Verifier checks these via oracle access or openings.

2.17.5.3 Final Verification via Polynomial Commitments

The GKR protocol concludes by reducing the initial fractional sumcheck claim to evaluation claims on the base multilinear extensions ($t, m, w_i, \ldots$) at a final random point, let's call it $\vec{r}_{final}$, derived from the verifier's challenges throughout the interaction.

To ensure the prover provides the correct evaluations at $\vec{r}_{final}$ without the verifier needing the full polynomials, a Polynomial Commitment Scheme (PCS) is used. This final step works as follows:

1. Commit Phase (Upfront): Before the GKR protocol starts, the prover commits to the table polynomial $t(\vec{X})$, the witness polynomials $w_i(\vec{X})$, and the newly constructed

doi:10.6342/NTU202503318

multiplicity polynomial $m(\vec{X})$. These commitments are sent to the verifier.

2. Opening Phase (Conclusion): After the GKR interaction, the prover provides the claimed evaluations of these polynomials at the random point $\vec{r}_{final}$. The prover then uses the PCS to generate a single, batched opening proof, π_{open}, for all these evaluations.

3. Verification: The verifier checks the opening proof π_{open} against the initial commitments and the claimed evaluations. If the check passes, the verifier is convinced that the evaluations are correct and thus accepts the original proof.

This is the crucial step that connects the abstract GKR interaction to the concrete polynomials of the lookup argument. The efficiency gain highlighted in LogupGKR is that the prover only needs to commit to one extra column—the multiplicities $m(\vec{X})$—compared to the original problem statement (which already includes t and w_i). This avoids the need for numerous helper columns found in other lookup arguments.

2.17.5.4 Costs and Performance Characteristics

- Prover Asymptotics: Arithmetic cost for GKR fractional sumcheck is approx. $|\mathbb{H}^{n+k'}| \cdot (43 \text{ Mult} + 29 \text{ Add})$. Critically, only one additional commitment is required: the multiplicity column $m(\vec{X})$.

- Sublinear in Table Size N?: Yes, as it avoids large table-related commitments if $M \ll N$.

- Preprocessing: No table-specific preprocessing is needed beyond standard PCS setup.

doi:10.6342/NTU202503318

- Proof Size (GKR): Typically small, related to $\log$(circuit size).
- Verifier Work (GKR): Efficient, logarithmic in circuit size.
- Key Benefit: Drastically reduced commitment cost compared to traditional LogUp.

Benchmarks show LogupGKR can be extremely fast in practice for both prover and verifier, outperforming pairing-based methods, especially when preprocessing is undesirable.

It is insightful to compare LogupGKR with CQ, as both leverage the logarithmic derivative technique. Their approaches to optimization, however, diverge significantly. LogupGKR employs the GKR protocol, whose primary advantage lies in its iterative verification process where intermediate layers do not require separate cryptographic commitments, thus substantially reducing commitment overhead. In contrast, CQ adopts a strategy of extensive preprocessing. It precomputes and caches commitments for quotients corresponding to every entry in the lookup table. This allows for the efficient composition of the final quotient polynomial commitment (Q_A) during the proving phase, which also saves significant commitment costs, but at the expense of a potentially massive and costly preprocessing step.

2.17.5.5 Generalizations and Variants

Univariate Extension: LogupGKR can be applied to univariate polynomials by transforming univariate commitments to multilinear ones via an IOP, using bit-decomposition of the univariate domain. The core GKR fractional sumcheck remains similar.

doi:10.6342/NTU202503318

2.17.6 Lasso ([6])

Lasso [6] provides a framework for *Indexed Lookup Arguments*, proving $f_i = t_{a_i}$, and techniques for large tables, often by exploiting table structure.

2.17.6.1 Core Concepts and Variants

An Indexed Lookup Argument proves for query f, index a, table t:

$$\forall i \in [0, m-1), f_i = t_{a_i} \tag{2.17.23}$$

Lasso presents several approaches:

1. Offline Memory Checking: Models lookups as VM reads.
2. Spark (Sparse Polynomial Commitments): Efficient PCS for sparse selector matrices.
3. Surge (Decomposable Tables): Leverages Spark for tables that decompose into smaller sub-tables.
4. Generalized Lasso (MLE-Structured Tables): For tables where entries t_i can be computed efficiently from index i.

2.17.6.2 Offline Memory Checking

Proves lookup by verifying a VM memory access log. Ensures multiset equality: $S_0 \cup \{W_j\} = S_m \cup \{R_j\}$, where S_0 is initial memory (table t, counters 0), $R_j = (a_j, f_j, c_j)$ are

doi:10.6342/NTU202503318

read logs, $W_j = (a_j, f_j, c_j+1)$ are write logs (counter increment), and S_m is final memory state. Verified via Grand Product Argument, which compares randomized fingerprints of the multisets.

2.17.6.3 Spark (Sparse Polynomial Commitments)

For sparse MLE $g(X)$ with m non-zero entries h_j at $k_j = (k_{j,x}, k_{j,y})$ (2D example). To prove $g(u) = v$:

$v = \sum_{j=0}^{m-1} h_j \cdot \mathsf{eq}(k_{j,x}, u_x) \cdot \mathsf{eq}(k_{j,y}, u_y)$. Let $e_j^{(x)} = \mathsf{eq}(k_{j,x}, u_x)$, etc.

The sum $v = \sum h_j e_j^{(x)} e_j^{(y)}$ is verified by sumcheck, reducing to point evaluations of $h, e^{(x)}, e^{(y)}$. Correctness of $e^{(x)}, e^{(y)}$ (i.e., $e_j^{(x)} = \mathsf{eq}(k_{j,x}, u_x)$) is proven by Offline Memory Checking against implicit tables defined by $\mathsf{eq}(\cdot, u_x)$ and $\mathsf{eq}(\cdot, u_y)$. Prover cost is $O(m + cN^{1/c})$ where c is a tunable parameter.

2.17.6.4 Surge (Decomposable Tables)

Applies to $f_i = t_{a_i}$ where $t_k = \mathcal{G}(t^{(0)}[\mathsf{dim}^{(0)}(k)], \ldots, t^{(c-1)}[\mathsf{dim}^{(c-1)}(k)])$.

The relation $\sum_b M(r, b) t(\mathrm{col}(b)) = f(r)$ is transformed using decomposition:

$f(r) = \sum_b \mathcal{G}(e^{(0)}(b), \ldots, e^{(c-1)}(b)) \cdot \mathsf{eq}(b, r)$, where $e^{(s)}(b) = t^{(s)}[\mathsf{dim}^{(s)}(\mathrm{col}(b))]$.

Verified by sumcheck, reducing to point evaluations of $e^{(s)}$. Correctness of each $e^{(s)}$ proven by Offline Memory Checking against sub-table $t^{(s)}$.

doi:10.6342/NTU202503318

2.17.6.5 Generalized Lasso (MLE-Structured Tables)

For tables where t_i is cheaply computable from i. Verifies $\sum_y M(r, y)t(y) = f(r)$ using a "Sparse-Dense Sumcheck". This sumcheck is optimized for sparse $M(r, \cdot)$ and efficiently evaluable $t(\cdot)$. Reduces to point evaluations $M(r, \rho)$, $t(\rho)$. $M(r, \rho)$ verified using Spark. $t(\rho)$ verified by PCS opening or direct computation by Verifier.

2.17.6.6 The Protocol (Conceptual Flow for Variants)

Specific protocols vary, but generally involve:

1. Commitments: Prover commits to queries f, indices a (or selector matrix M), and any auxiliary vectors (e.g., $e^{(s)}$ in Surge, counters in OMC). Table t (or sub-tables) may be pre-committed or, if structured, implicitly defined and not committed to at all.

2. Challenges: Verifier sends random challenges for sumchecks, combinations, evaluation points.

3. Sumchecks / Reductions: Core relations are reduced via sumcheck protocols (standard, sparse-dense, or GKR-like).

4. Point Evaluations & Openings: Prover provides evaluations of polynomials at challenged points, along with PCS proofs.

5. Recursive Proofs (if any): Some components like Spark or Surge use Offline Memory Checking internally, which itself is another lookup-like argument.

doi:10.6342/NTU202503318

2.17.6.7 Costs and Performance Characteristics

Lasso's goal is often to make prover costs dependent on query count m rather than large table size N, especially for structured tables.

- Offline Memory Checking: Prover cost $O(m + N)$ for constructing logs and polynomials. Grand Product argument varies.
- Spark: Prover cost for eval argument $O(m + cN^{1/c})$ (with c-dim decomposition).
- Surge: Depends on sub-table sizes and complexity of $\mathcal{G}$. Aims for $O(m + \sum |t^{(s)}|)$.
- Generalized Lasso: Prover cost for Sparse-Dense Sumcheck can be $O(cm)$ if $N = m^c$ and t has the required structure for "condensation".
- Key Idea: Leverage table structure (decomposition, MLE-structure) or query sparsity.

2.17.6.8 Generalizations and Variants

Lasso offers a suite of tools. Decomposable tables include RangeCheck, bitwise operations (AND, OR, XOR), equality (EQ), less-than (LTU), shifts (SLL), each with specific decomposition functions $\mathcal{G}$ and sub-tables. The choice of protocol depends on table properties.

Summary of Evolution

1. From $O(N)$ to Sublinear(N): Caulk made the crucial leap by shifting the workload from the entire table to a small, extracted sub-table.

doi:10.6342/NTU202503318

2. From $O(m^2)$ to $O(m \log m)$: The Baloo $\rightarrow$ cq evolution fixed the new bottleneck introduced by Caulk, making the cost nearly linear in the number of lookups. cq further refined this by enabling efficient proof aggregation.

3. From Table Commitment to Table-Free: Lasso changed the game entirely. Instead of proving inclusion in a committed table, it proves a sparse relationship. For structured tables, it removes the table commitment dependency altogether, enabling massive tables and making the prover's cost dependent only on the accessed data, not the entire potential lookup space.

Benchmarking Related Literature Review

As Zero-Knowledge Proofs (ZKPs) transition from theoretical constructs to practical applications, the need for robust performance evaluation and benchmarking of different proof systems has become critical [49, 50]. However, the landscape has been characterized by a lack of standardized testing frameworks, reproducible results, and uniform evaluation metrics, creating significant challenges for developers in selecting optimal solutions for their specific use cases [49]. In response, recent academic and community efforts have produced a body of work focused on benchmarking, spanning comprehensive frameworks, platform-specific comparisons, and application-oriented optimizations.

Comprehensive Benchmarking Frameworks and Comparative Studies

To address the absence of standardized evaluation in the ZKP ecosystem, researchers have developed holistic benchmarking frameworks. zk-Bench stands out as the first com-

doi:10.6342/NTU202503318

prehensive benchmarking framework and estimator tool designed specifically for general-purpose ZKP systems, particularly SNARKs [49]. Its evaluation scope ranges from low-level cryptographic arithmetic libraries to high-level ZKP circuits, with the goal of providing reproducible and fair comparative data [49]. The analysis from zk-Bench revealed that the performance of ZKP tools varies significantly with hardware, showing performance gains of up to 50% on CPU-optimized machines and 40% on memory-optimized machines, depending on the tool [49]. Furthermore, the study offers a detailed quantitative comparison of the setup, proving, and verification phases of different proof systems, such as Groth16 and Plonk [49].

In addition to new frameworks, systematic reviews have provided high-level comparisons of major ZKP technologies. The work by El-Hajj et al., for instance, evaluates the efficiency of zk-SNARKs, zk-STARKs, and Bulletproofs in real-world scenarios [51]. Their findings conclude that zk-SNARKs produce the smallest proofs, while zk-STARKs generate the largest proofs but are the fastest in proof generation and verification times; Bulletproofs were found to be the slowest in both aspects [51]. Such studies offer developers a high-level overview of the inherent trade-offs between different ZKP families.

Platform-Specific and Application-Specific Benchmarking

Beyond general frameworks, a significant body of work focuses on evaluating performance within specific platforms or for particular applications. The zk-benchmarking project, for example, is a suite designed to compare ZK virtual machines (ZK-VMs), currently benchmarking STARK-based systems like Polygon Miden and RISC Zero [50]. This initiative emphasizes a set of core principles, ensuring its benchmarks are relevant,

doi:10.6342/NTU202503318

neutral, idiomatic, and reproducible [50]. Its test cases include critical building blocks for real-world applications, such as iterated hashing and Merkle inclusion proofs [50].

Other research provides deep-dive benchmarks for specific development ecosystems. Steidtmann et al. presented comprehensive benchmarking results for various signature schemes and hash functions implemented in Circom [52]. This work aids developers working within the Circom environment by providing concrete performance insights to guide the selection of appropriate cryptographic schemes for their applications [52].

Furthermore, research has also targeted the optimization and evaluation of specific components for target environments like blockchains. Guo et al. focused on benchmarking ZK-friendly hash functions, such as Poseidon2, for EVM-compatible blockchains [53]. Their results demonstrated that using Poseidon2 can reduce on-chain costs by 73% on EVM chains and improve proof generation times, thereby enhancing privacy and efficiency in ZKP-based protocols [53]. This component-level benchmarking is crucial for engineering high-performance and cost-effective ZKP applications.

doi:10.6342/NTU202503318

Chapter 3 Design and Experiment

3.1 Implementation Framework and Reference Implementations

My implementation is based on the `plonkish` framework [54], which provides the building blocks for constructing zkSNARKs, such as MSM, FFT, transcript, and sum-check. A complete list of these components can be found in the `plonkish_backend` [55]. However, the lookup argument itself is not included in this framework, so I needed to implement it myself.

I referenced the following implementations and integrated the concepts into the `plonkish` framework. Due to significant differences in the implementation approaches of each repository, I essentially had to write the code from scratch after understanding the core concepts.

- Plookup: The production-grade implementation in `halo2` [56].
- Caulk: The research-grade implementation in [57].
- Baloo: No known implementation.

doi:10.6342/NTU202503318

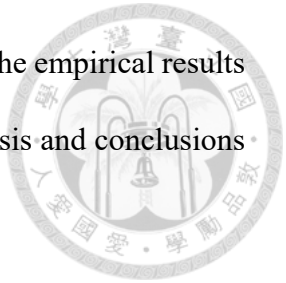
- CQ: The research-grade implementation in [58].
- LogupGKR: No known implementation, but fractional sumcheck is implemented in [59].
- Lasso: This is highly coupled with Jolt, a zkVM. The implementation is available at [60], but it is tightly coupled with Jolt and could not be used directly. I had to rewrite it. Fortunately, DoHoonKim8 wrote `halo2-lasso` [61], and I based my implementation on their work, modifying some of the `plonkish` code to make it run. The details of my modifications can be found in this commit [62]. After these modifications, I was able to integrate it and run it.

In the preceding chapters, we explored the theoretical constructions and complexities of various lookup arguments. However, theoretical analysis alone provides only a high-level comparison of asymptotic behavior. Real-world performance is invariably influenced by implementation-specific optimizations, underlying cryptographic libraries, and the specific hardware environment. To conduct a comprehensive and equitable evaluation of the leading schemes—from foundational protocols like Plookup [1] to the latest sublinear-N advancements such as Caulk [2], Baloo [3], CQ [4], Lasso [6], and LogupGKR [5]—this section details our experimental design, implementation framework, and empirical results.

The central objective is to systematically quantify and compare the performance of these lookup arguments across four key metrics: Proving Time, Verification Time, Setup and Preprocessing Cost, and Proof Size. In this section, we will first introduce the implementation framework that forms the basis of our evaluation. Subsequently, we will elaborate on the design of our benchmarking scenarios, which are crafted to highlight the

doi:10.6342/NTU202503318

strengths and weaknesses of each scheme under various conditions. The empirical results presented in next section will provide a solid foundation for the analysis and conclusions that follow.



3.2 Integration of Heterogeneous Lookup Arguments

The core challenge in evaluating lookup arguments lies in their heterogeneous mathematical tools, data structures, and proof processes. This section details our unified benchmarking framework that integrates these diverse schemes into a single executable testing environment.

3.2.1 Challenge: Heterogeneous Interfaces and Data Models

Each scheme adopts data representations optimized for its theoretical model, creating significant integration challenges:

3.2.1.1 Different Input Data Structures

Schemes such as Plookup [1], CQ [4], and Caulk [2] accept two vectors as input: a lookup vector `lookup: Vec<Fr>` and a table vector `table: Vec<Fr>`. Baloo's [3] implementation requires a more complex internal structure, directly handling polynomials and their evaluations at specific points, and constructing multiple auxiliary polynomials (such as $v(X)$, $D(X)$, $E(X)$, etc.) during the proof process. LogupGKR's [5] theory is based on logarithmic derivatives and the GKR protocol [46], with its direct input being multiple multisets. This requires converting traditional lookup and table structures into its

doi:10.6342/NTU202503318

specific format, including calculating the multiplicities of each table element. Lasso [6] is based on sparse polynomial commitments, with its Prover focusing on the non-zero entries of the lookup matrix M, and its input structure also differs from other schemes.

3.2.1.2 Differences in Proof Processes and Parameter Generation

Setup Dependencies: PCS-based schemes (Plookup [1], Caulk [2], Baloo [3]) require KZG SRS generation, while LogupGKR [5] and Lasso [6] employ alternative setup procedures.

3.2.2 Integration and Abstraction of Underlying Libraries

3.2.2.1 PlonkishBackend Trait

In our codebase, the `PlonkishBackend` trait defines a more generic interface, including standard lifecycle methods such as `setup`, `preprocess`, `prove`, and `verify`. This allows solutions like Caulk [2] to be implemented as an instance of this trait.

```
impl<M> Caulk<M> {
    // These functions constitute the specific interface of the
       Caulk solution
    pub fn setup(...) -> ...
    pub fn prove(...) -> ...
    pub fn verify(...) -> ...
}
```

Listing 3.1: Caulk Implementation Structure

doi:10.6342/NTU202503318

3.2.2.2 Abstractions for Polynomial Commitment Schemes

This modular design enables future PCS replacements (e.g., IPA [24] or FRI) through simple type alias modifications.

```
// Bind Plookup to a specific PCS via type alias
type PlookupBn256 = plookup::Plookup<Fr, UnivariateKzg<Bn256>>;
```

Listing 3.2: PCS Abstraction

3.2.3 Shared Cryptographic Components for Fair Benchmarking

To achieve rigorous apple-to-apple comparisons, our benchmarking framework employs a unified set of cryptographic primitives that eliminates implementation-specific performance variations. This design ensures that observed performance differences stem from protocol design choices rather than underlying library disparities.

3.2.3.1 Polynomial Commitment Scheme Decoupling

The framework implements a sophisticated trait-based decoupling mechanism that separates proving protocols from their underlying PCS implementations. Located in `plonkish_backend/src/backend/`, each protocol's Prover and Verifier are implemented as generic structures parameterized by a `PolynomialCommitmentScheme` trait.

All protocols utilize identical KZG implementations on `halo2_curves::bn256`, ensuring performance variations reflect protocol design rather than implementation differences.

doi:10.6342/NTU202503318

3.2.3.2 Unified Sum-Check Protocol

The shared Sum-Check implementation (`plonkish_backend/src/piop/sum_check/classic.rs`) provides a standardized foundation for multilinear polynomial verification. This component is particularly crucial for protocols like LogupGKR [5] and Lasso [6], which rely heavily on Sum-Check operations as their computational core. The unified implementation ensures that these protocol comparisons reflect algorithmic design differences rather than Sum-Check implementation variations.

3.2.3.3 Standardized Arithmetic Operations

Two fundamental arithmetic operations are standardized across all implementations:

Multi-Scalar Multiplication (MSM): Located in `plonkish_backend/src/util/arithmetic/msm.rs`, this component handles the computationally intensive elliptic curve operations $\sum_{i=1}^{n} s_i \cdot g_i$. Since KZG commitments fundamentally reduce to MSM operations, this standardization ensures equitable treatment of all polynomial commitment-dependent protocols.

Fast Fourier Transform (FFT): The implementation (`plonkish_backend/src/util/arithmetic/fft.rs`) employs an optimized iterative Radix-2 Cooley-Tukey algorithm with several performance enhancements:

- In-place computation minimizing memory allocation overhead
- Pre-computed evaluation domains with cached roots of unity
- Bit-reversal permutation optimization for improved cache locality

doi:10.6342/NTU202503318

- Coset FFT support required for quotient polynomial calculations

These optimizations ensure that polynomial multiplication and division operations—fundamental to all lookup protocols—operate at consistent efficiency levels across different schemes.

3.2.3.4 Fiat-Shamir Transcript Standardization

The unified transcript implementation (`plonkish_backend/src/util/transcript.rs`) standardizes the non-interactive transformation process. All protocols utilize identical sponge construction algorithms for generating challenge scalars (α, β, γ, etc.), ensuring consistent security assumptions and eliminating transcript-related performance variations from the comparison.

3.2.4 Experimental Framework and Design

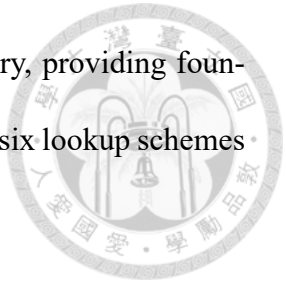
3.2.4.1 Implementation Framework

Our evaluation is based on a unified, high-performance Rust implementation framework designed for benchmarking cryptographic protocols. This framework is available at nooma-42/Lookup-Argument [1].

Hardware Environment: All benchmarks were conducted on an Apple M4 system equipped with a 10-core CPU, 10-core GPU, 24GB unified memory, and 512GB SSD storage. This configuration provides consistent computational resources and eliminates hardware-induced performance variations across different protocol evaluations.

[1] https://github.com/nooma-42/Lookup-Argument

doi:10.6342/NTU202503318



- Core Architecture: Built upon the `plonkish` research repository, providing foundational polynomial operations and PIOP components [28]. All six lookup schemes are integrated within this unified architecture.
- Parallel Execution Engine: To accelerate comprehensive testing, the framework incorporates a two-tiered parallel execution model using the `rayon` crate.
  - Benchmark-Level Parallelism: Multiple benchmark configurations (combinations of system, table size, and lookup ratio) are executed concurrently across available CPU cores.
  - Algorithm-Level Parallelism: Specific algorithms, such as Lasso and LogupGKR, whose internal structures are amenable to parallelization, are implemented to leverage this feature. This provides a practical lens through which to view theoretical algorithm design, as schemes with inherently parallelizable structures can achieve significant real-world speedups on multi-core hardware.
- Standalone Nature: It is important to note that the current implementations of these lookup arguments function as standalone modules within the Plonkish framework. They are not yet directly integrated with a complete front-end constraint system like Halo2 [13], but are designed to benchmark the core cryptographic primitives in a controlled environment.

3.2.4.2 Evaluation Metrics and Scenario Design

To comprehensively evaluate the performance of the different lookup arguments, we designed a series of benchmarking scenarios focused on the four critical metrics. The benchmark parameters are primarily controlled by `K`, which defines the table size

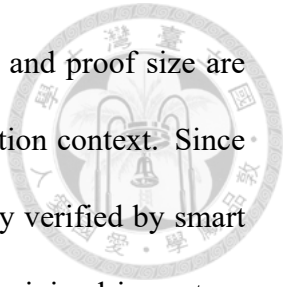
doi:10.6342/NTU202503318

as $N = 2^K$, and `ratio`, which defines the lookup size as $n = N/\text{ratio}$. In our comparative analysis, we unify the benchmark task for all lookup arguments to that of a range check. The primary rationale for this standardization is to accommodate the architectural design of the Lasso protocol. Lasso's efficiency is contingent upon the lookup table being "decomposable"—that is, expressible as a function of smaller, independent subtables. A canonical decomposable function for range checks is well-established, making it a practical use case for Lasso. Conversely, lookups on arbitrary, unstructured datasets do not possess a readily available decomposable structure, rendering them incompatible with Lasso's core mechanism. Therefore, standardizing on range checks creates a level playing field, enabling a direct performance evaluation of Lasso alongside protocols like CQ and Plookup.

Prover Time Analysis under Varying Parameters Prover time is often the most critical bottleneck in ZK-SNARK systems. Our analysis is designed to validate the theoretical complexities of each protocol by observing its response to changing parameters.

- Impact of Table Size (N) and Lookup Size (n): We measure prover time while varying `K` (from 4 to 13) and `ratio` (from 2 to 16) to systematically test each protocol's scaling behavior. Based on theoretical analysis, we expect Plookup to exhibit $O(N)$ degradation due to its linear dependency on table size, while sublinear-N protocols should scale primarily with n rather than N. This experimental design allows us to isolate and validate key complexity differences: Caulk's quadratic $O(n^2)$ bottleneck versus the more efficient $O(n \log n)$ behavior of Baloo and CQ, while confirming that Lasso and LogupGKR remain largely insensitive to lookup size variations.

doi:10.6342/NTU202503318



Verification Time and Proof Size Analysis While verification time and proof size are often considered critical metrics, they are less decisive in our evaluation context. Since all evaluated protocols produce SNARK proofs that can be efficiently verified by smart contracts, the practical differences in verification performance have minimal impact on real-world deployment scenarios. Nevertheless, we empirically measure and present these metrics to provide a complete performance profile and identify any notable variations among the schemes.

Setup and Preprocessing Costs The one-time setup cost is a crucial factor for applications, particularly those involving large, static tables.

- Setup Requirements: Protocols differ significantly in preprocessing needs—from $O(N \log N)$ table-dependent setup (CQ, Caulk) to minimal preprocessing (Lasso, LogupGKR), affecting their suitability for different application scenarios.

3.2.4.3 Data Collection and Analysis

The benchmarking framework is designed to systematically collect performance data across all parameter combinations. The results are then aggregated and visualized to facilitate a clear comparative analysis.

- Data Logging: The framework logs all performance metrics (prover time, verifier time, proof size, setup time) for each benchmark run.
- Visualization: The collected data is used to generate plots that illustrate the performance trends of each protocol as a function of table size and lookup ratio. These

doi:10.6342/NTU202503318

visualizations are essential for interpreting the results and drawing meaningful conclusions.

By combining a unified implementation framework with a carefully designed set of benchmarking scenarios, we can provide a comprehensive and equitable evaluation of the leading lookup argument schemes. The results of this evaluation will offer valuable insights into the practical trade-offs of each protocol and guide the selection of the most appropriate scheme for different application requirements.

doi:10.6342/NTU202503318

Chapter 4 Evaluation and Discussion

4.1 Performance Analysis and Visualization

This section presents a comprehensive performance analysis through four carefully designed visualizations that collectively demonstrate the evolution and comparative advantages of different lookup argument systems. These figures provide empirical validation of theoretical complexity analyses and reveal critical insights into the practical implications of algorithmic design choices.

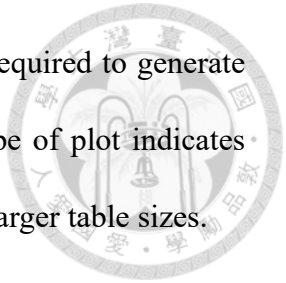
4.1.1 Overall System Performance Comparison

4.1.1.1 Graph Interpretation

Each graph plots the time it takes to generate a proof against the size of the lookup table.

X-Axis (Lookup Table Size K): This represents the size of the data table being looked up. The size is exponential, calculated as 2^K. So, as K increases from 5 to 13, the table size grows dramatically.

doi:10.6342/NTU202503318



Y-Axis (Proving Time): This is the time in milliseconds (ms) required to generate the proof, shown on a logarithmic scale. A straight line on this type of plot indicates exponential growth. Flatter lines mean the system scales better with larger table sizes.

Lines: Each colored line represents a different lookup protocol being tested (Baloo, CQ, Caulk, Lasso, LogupGKR, Plookup). These are the systems implemented in the `plonkish_backend` of the Rust project.

N:n Ratio: Each of the four graphs is generated with a different "N:n ratio" (2, 4, 8, and 16). This ratio compares the number of lookups performed (N) to the size of the table (n). A higher ratio means more lookups are being done relative to the table's size.

4.1.1.2 Performance Analysis

Across all four graphs, a clear performance pattern emerges:

Top Performers: Lasso and LogupGKR are consistently the fastest systems by a large margin. Their proving times are significantly lower and increase much more slowly as the table size grows. This superior scalability is visible in their relatively flat lines on the graphs.

Mid-Tier Performers: Plookup, Baloo, and CQ form a middle group. Their performance is much slower than Lasso and LogupGKR, and they scale less efficiently with larger tables, as shown by their steeper curves.

Slowest Performer: Caulk is consistently the slowest protocol. Its proving time increases very rapidly with table size, making it the least scalable option among those tested.

doi:10.6342/NTU202503318

4.1.1.3 Effect of the N:n Ratio

By comparing the four charts, we can see how the number of lookups affects performance:

As the N:n ratio increases from 2 to 16, the proving time for all systems increases. This is expected, as a higher ratio means the system must do more work.

The relative performance ranking does not change. Lasso and LogupGKR remain the fastest, and Caulk remains the slowest, regardless of the ratio.

The performance gap between the top performers and the rest of the systems becomes even more pronounced at higher ratios.

In summary, these benchmarks clearly demonstrate that for the range of parameters tested, Lasso and LogupGKR offer substantially better performance and scalability for proving lookups compared to the other systems, as shown in Figure 4.1.

4.1.1.4 Baloo Discrepancy and Caulk Implementation Bottleneck

The benchmark results reveal specific implementation-related performance issues that warrant detailed examination, particularly regarding Baloo's validation discrepancy and Caulk's implementation bottleneck that significantly impact the overall system performance comparison.

doi:10.6342/NTU202503318

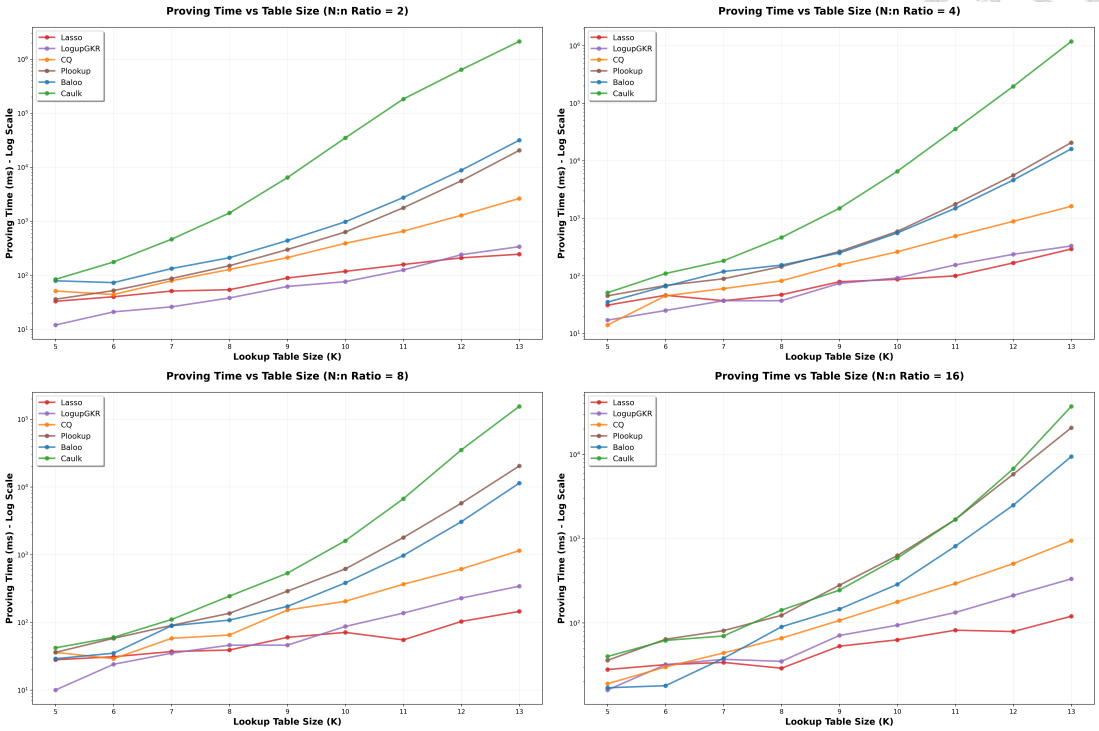


Figure 4.1: Prover time versus lookup table size K for different N:n ratios (2, 4, 8, and 16 from top-left to bottom-right). All graphs use logarithmic scales on the y-axis. The lines represent different lookup argument systems: Baloo (blue), CQ (orange), Caulk (green), Lasso (red), LogupGKR (purple), and Plookup (brown).

4.1.1.5 Crossover Analysis: Lasso vs. LogupGKR

While both Lasso and LogupGKR are top performers, they are not identical. A distinct crossover pattern reveals how their relative performance changes based on the lookup table size (K) and the lookup frequency ($N : n$ ratio).

The general trend is that LogupGKR is often slightly faster for very small table sizes, but Lasso quickly overtakes it and becomes the faster protocol as the table size increases.

Here is a breakdown of the crossover point in each graph:

N:n Ratio = 2:

- For small tables ($K = 5, 6$), LogupGKR (purple) has a slight performance advantage.

doi:10.6342/NTU202503318

- The crossover occurs between K=6 and K=7.
- For $K \geq 7$, Lasso (red) is consistently faster.

N:n Ratio = 4:

- LogupGKR is faster at K=5.
- The two are nearly identical in performance at K=6 and K=7.
- The crossover happens around K=7.
- For $K \geq 8$, Lasso establishes a clear performance lead.

N:n Ratio = 8:

- LogupGKR is faster only at the smallest table size, K=5.
- The crossover happens early, between K=5 and K=6.
- For all $K \geq 6$, Lasso is the faster system.

N:n Ratio = 16:

- Similar to the previous chart, LogupGKR is only faster at K=5.
- The crossover again occurs between K=5 and K=6.
- For all subsequent table sizes, Lasso is faster.

doi:10.6342/NTU202503318

4.1.1.6 Interpretation of the Trend

The key takeaway is that the crossover point shifts to a smaller table size as the $N : n$ ratio increases. This suggests that Lasso's architecture scales more effectively not just with table size, but also with the lookup density (the number of lookups relative to the table size). When an application requires a large number of lookups (a high $N : n$ ratio), Lasso's performance advantage becomes apparent even with smaller tables.

Lasso and LogupGKR Consistency: Both systems maintain exceptional stability across all tested values of n, confirming their theoretical independence from lookup count. While LogupGKR may be marginally faster for very small tables with low lookup frequency, Lasso demonstrates superior scalability for larger tables or high-frequency lookup scenarios.

Plookup's Stability: Plookup (brown) exhibits a nearly straight trajectory, demonstrating that with fixed table size N, its $O(N + n)$ complexity is dominated by the $O(N)$ constant term and remains insensitive to variations in n. While Plookup's absolute performance is slow, its predictable behavior represents a significant advantage in scenarios with varying lookup requirements.

Baloo and CQ Success: Both Baloo (blue) and CQ (orange) exhibit nearly straight curves similar to Plookup, indicating successful reduction of n-dependency from quadratic to near-linear complexity. Even as the number of lookups n increases substantially, their performance remains stable, completely resolving Caulk's bottleneck.

In conclusion, while both top-performing systems are excellent, the choice depends on the specific use case:

doi:10.6342/NTU202503318

- For applications with very small tables and a low number of lookups, LogupGKR may be marginally faster.
- For applications involving larger tables or a high frequency of lookups, Lasso demonstrates superior scalability and is the more efficient choice.

4.1.1.7 Validation of Caulk's Implementation Bottleneck

Figure 4.2 provides the most insightful analysis in our study by isolating the impact of lookup count n while maintaining a fixed large table size.

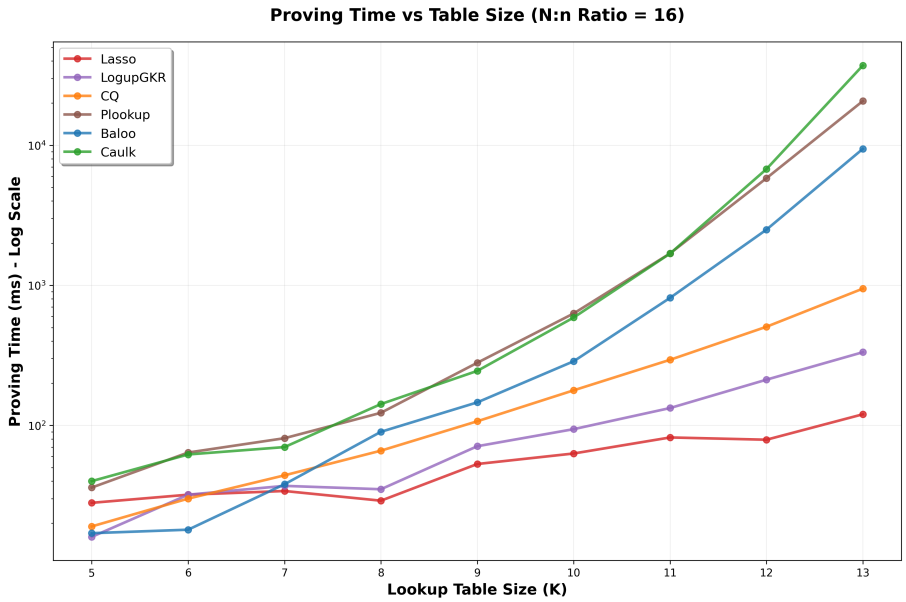


Figure 4.2: Prover time versus lookup count n with fixed table size $K = 11$ ($N = 2048$). Both axes use logarithmic scales. The lookup count n is varied by adjusting the $N : n$ ratio parameter.

Caulk's Proving Time Quadratic Complexity Validation: The Caulk system (green) exhibits an extremely steep upward trajectory. In the double-logarithmic coordinate system, the theoretical slope for $y = x^2$ complexity is 2, while linear complexity $y = x$ exhibits a slope of 1. Caulk's observed slope significantly exceeds all other systems, providing clear visual confirmation of its $O(n^2)$ complexity. When the lookup count n in-

doi:10.6342/NTU202503318

creases from 128 to 1024, the execution time escalates from hundreds of milliseconds to tens of seconds, demonstrating the undesirable nature of this scaling behavior.

4.1.1.8 Baloo Discrepancy

Empirical Observation: For both the Baloo and CQ systems, modifications to the ratio parameter (corresponding to changes in n) demonstrate minimal impact on prover time when the parameter K remains fixed.

Theoretical Framework: According to theoretical analysis, Baloo's prover time complexity is $O(m \log^2 m)$, while CQ achieves $O(m \log m)$ complexity, where m corresponds to the parameter n in our experimental setup.

Analysis of Theoretical-Experimental Discrepancies:

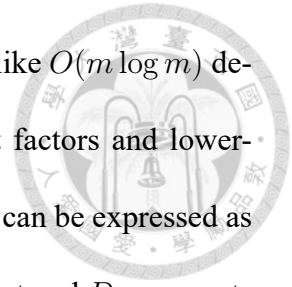
From a purely theoretical perspective, prover time should exhibit a decreasing trend as n decreases. Specifically, when $K = 10$ remains fixed, varying the ratio from 2 to 16 (corresponding to n changing from 2^9 to 2^6) should result in observable reductions in prover time due to the logarithmic dependency on m.

However, experimental data for Baloo reveals that when $K = 10$, the recorded prover times are 1012ms, 1002ms, 1025ms, and 984ms, respectively. These measurements fail to demonstrate a clear decreasing trend and instead exhibit fluctuations that appear to be within measurement uncertainty bounds.

Potential Explanations for Observed Discrepancies:

Several factors may contribute to the apparent discrepancy between theoretical predictions and experimental observations:

doi:10.6342/NTU202503318



- Constant Factor Dominance: Asymptotic complexity notation like $O(m \log m)$ describes long-term behavior but does not account for constant factors and lower-order terms. In practical implementations, total execution time can be expressed as $C \cdot f(m) + D$, where C is the complexity-dependent coefficient and D represents fixed overhead costs. When the parameter range is relatively constrained (e.g., 2^6 to 2^9) or when the constant term D constitutes a significant proportion of the total execution time, variations in the $C \cdot f(m)$ component may be masked by measurement noise and constant overhead.

- Implementation-Specific Overheads: Practical implementations may incorporate memory allocation strategies, initialization procedures, or other system-level operations designed to handle varying values of n. These implementation details can introduce fixed overhead costs that remain constant across different parameter values, thereby obscuring the theoretical scaling behavior.

4.2 Setup Time Performance Analysis

The setup phase represents a critical component in the practical deployment of lookup argument systems. This section presents a comprehensive analysis of setup time performance across different lookup protocols under varying lookup density conditions.

4.2.1 Experimental Setup and Methodology

The setup time analysis employs four distinct N:n ratios (2, 4, 8, and 16) to evaluate protocol performance under different lookup density scenarios. The y-axis represents

doi:10.6342/NTU202503318

setup time in milliseconds using a logarithmic scale, while the x-axis indicates lookup table size K, where the actual table size is $N = 2^K$. On logarithmic plots, a linear relationship indicates exponential growth, corresponding to linear time complexity $O(N)$ with respect to table size.

4.2.2 Protocol Classification and Performance Characteristics

The experimental results reveal a clear bifurcation of protocols into two distinct performance categories based on their fundamental algorithmic approaches:

4.2.2.1 Linear Setup Time Protocols ($O(N)$ Complexity)

This category encompasses protocols that require comprehensive preprocessing of the entire lookup table, resulting in setup times that scale linearly with table size N.

CQ and Caulk: These protocols exhibit the highest setup overhead within this category. Their setup procedures involve expensive precomputational operations across the entire table N, resulting in substantial computational costs. The linear scaling behavior is clearly visible as straight-line trajectories on the logarithmic plots.

Plookup and Baloo: While still exhibiting linear scaling with table size N, these protocols demonstrate improved constant factors compared to CQ and Caulk. Their setup times remain predictably proportional to table size, but with reduced algorithmic constants that translate to better practical performance.

Insensitivity to Lookup Density: A critical characteristic of this category is that setup time remains relatively unchanged across different N:n ratios. Since these protocols must

doi:10.6342/NTU202503318

process the entire table N regardless of the number of lookups required, variations in lookup density (n) have minimal impact on setup performance.

4.2.2.2 Sub-linear Setup Time Protocols ($O(n)$ Complexity)

This category represents a paradigmatic shift in lookup argument design, where setup time scales primarily with the number of lookups rather than table size.

Lasso and LogupGKR: These protocols demonstrate exceptional setup efficiency, with their performance curves appearing nearly horizontal on logarithmic plots. This behavior indicates that setup time is predominantly determined by the number of lookups (n) rather than table size (N), as demonstrated in Figure 4.3.

Adaptive Scaling with Lookup Density: The most remarkable characteristic of these protocols is their sensitivity to the N:n ratio. As this ratio increases (indicating sparser lookups), their setup time curves become increasingly flat. In the N:n = 16 configuration, both Lasso and LogupGKR exhibit nearly constant setup times across all tested table sizes, demonstrating that their setup procedures successfully decouple from table size dependencies.

Practical Implications: This performance characteristic represents a fundamental advantage for applications requiring lookups from large tables with relatively few queries. The ability to achieve setup times independent of table size N enables practical deployment in scenarios previously considered computationally prohibitive.

doi:10.6342/NTU202503318

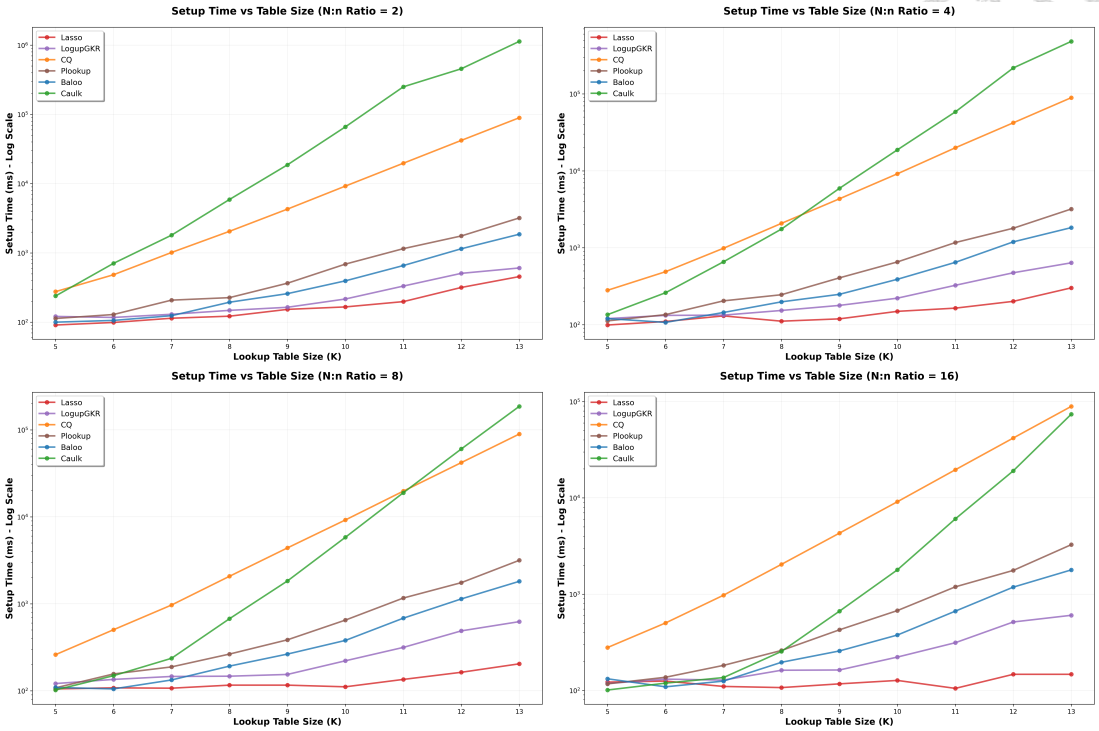


Figure 4.3: Setup time versus lookup table size K for different N:n ratios (2, 4, 8, and 16 from top-left to bottom-right). All graphs use logarithmic scales on the y-axis. The protocols demonstrate clear bifurcation into linear-time (CQ, Caulk, Plookup, Baloo) and sub-linear-time (Lasso, LogupGKR) categories.

4.3 Proof Size and Verification Time Analysis

While proving time and setup efficiency represent the primary performance bottlenecks in practical deployments of lookup argument systems, proof size and verification time constitute secondary considerations that merit examination for completeness. These metrics are generally of reduced importance due to their inherently constrained nature in modern zero-knowledge proof systems, where protocol design typically ensures that both proof sizes remain compact and verification procedures are efficient regardless of computational complexity.

doi:10.6342/NTU202503318

4.3.1 Proof Size Characteristics

The analysis of proof size reveals fundamental differences in the underlying cryptographic approaches employed by different protocol families.

4.3.1.1 GKR-Based Protocols (LogupGKR, Lasso)

GKR-based protocols employ an iterative reduction approach, analogous to peeling layers of an onion. The proof construction process systematically reduces large problems into smaller subproblems across $\log(N)$ layers, with each layer requiring verification of the correctness of the reduction step.

Dynamic Growth Pattern ($O(\log N)$): The proof size in these systems exhibits logarithmic growth with respect to table size. This scaling behavior arises because the proof must encode the intermediate results and verification data for each reduction layer. Consequently, larger lookup tables necessitate more reduction layers, resulting in proportionally larger proofs.

4.3.1.2 Permutation and Polynomial-Based Protocols (Plookup)

Permutation-based protocols adopt a transformation and compression paradigm. The entire lookup problem is transformed into a comprehensive algebraic identity, which is subsequently compressed using polynomial commitment techniques.

Constant Size Pattern ($O(1)$): The proof size in these systems remains constant regardless of table size, as illustrated in Figure 4.4. This property emerges because the final proof consists of a fixed number of cryptographic objects (commitments and opening

doi:10.6342/NTU202503318

proofs) whose size is independent of the underlying computation complexity. The polynomial commitment scheme effectively compresses arbitrary-sized computational proofs into constant-sized cryptographic certificates.

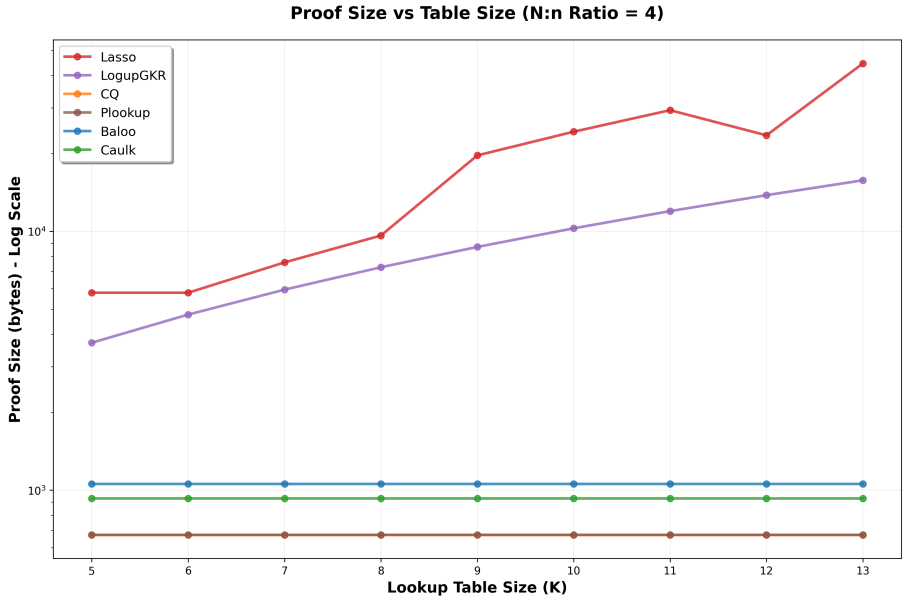


Figure 4.4: Proof size in bytes versus lookup table size K for N:n ratio of 4.0. The graph demonstrates the fundamental difference between GKR-based protocols (logarithmic growth) and permutation-based protocols (constant size).

4.3.2 Why Lasso's Proof Size Decreases at $K = 12$?

In our benchmark analysis of the Lasso protocol, we observed a non-monotonic relationship between the table size parameter, denoted by K, and the final proof size. While the proof size generally increases with K, a notable anomaly occurs at the transition from $K = 11$ to $K = 12$. Specifically, benchmark data from `benchmark_results_v2.csv` shows that for a fixed `N_to_n_Ratio` of 8, the proof size decreases from 24,256 bytes at $K = 11$ to 20,160 bytes at $K = 12$. This counter-intuitive result prompted a deeper investigation into the protocol's implementation.

Our analysis of the source code reveals that this phenomenon is not an error, but a

doi:10.6342/NTU202503318

deterministic outcome of Lasso's core table decomposition mechanism, which is highly sensitive to the arithmetic properties of K. The root cause lies in the `DecomposableTable` trait implementation for `RangeTable` within the file `plonkish_backend/src/backend/lasso.rs`. The `chunk_bits()` method in this implementation divides the K bits of a table index into smaller limbs of a fixed size, which is set to 4 in our benchmark context.

- For $K = 12$, which is a multiple of the limb size 4, the table index is uniformly decomposed into three 4-bit chunks: `[4, 4, 4]`. This results in three structurally identical sub-tables and their corresponding multilinear polynomials. The uniformity of this structure allows for highly efficient batch processing within the Polynomial Commitment Scheme (PCS), as all polynomials are defined over the same domain and can be treated homogeneously.

- For $K = 11$, which is prime, the decomposition is necessarily non-uniform, resulting in chunks of `[4, 4, 3]`. This heterogeneity forces the protocol to handle two different types of sub-tables (4-bit and 3-bit).

This structural asymmetry introduces additional complexity into the proof generation. Specifically, to create a single batch opening proof for polynomials of different sizes, the underlying cryptographic machinery must account for their different domains. This requires including additional structured information, in the form of `Evaluation<F>` instances, into the proof transcript to certify the consistency of operations across these heterogeneous domains. These additional `Evaluation<F>` instances, which contain field elements and metadata, are the concrete artifacts that increase the total size of the serialized proof. Therefore, the efficiency gained from a uniform decomposition at $K = 12$ outweighs the marginal increase in table size, leading to a smaller final proof compared to

doi:10.6342/NTU202503318

the less efficient, non-uniform decomposition required for $K = 11$.

4.3.3 Verification Time Analysis

The verification time analysis reveals performance characteristics that fundamentally differ from the proving time patterns observed in previous sections.

4.3.3.1 Table Size Independence

The most significant characteristic across all protocols is the independence of verification time from lookup table size K. The verification curves remain essentially flat across all tested table sizes, with observed variations primarily attributable to measurement noise rather than algorithmic scaling.

This behavior exemplifies a fundamental advantage of modern succinct non-interactive argument (SNARK) systems: verification cost remains constant or exhibits only logarithmic growth, regardless of the underlying computational complexity, as demonstrated in Figure 4.5. This property enables practical deployment scenarios where computationally intensive proofs can be verified efficiently by resource-constrained parties.

4.3.3.2 Protocol Performance Stratification

The verification time results demonstrate clear stratification into two distinct performance tiers:

High-Efficiency Verification Tier: Protocols such as Plookup, Baloo, and CQ demonstrate verification times in the range of 1-4 milliseconds across all configurations. This

doi:10.6342/NTU202503318

efficiency stems from their reliance on well-optimized polynomial commitment schemes and streamlined verification procedures.

Moderate-Efficiency Verification Tier: Lasso and LogupGKR exhibit verification times approximately one order of magnitude higher (10-40 milliseconds). This performance characteristic reflects the additional complexity introduced by their sophisticated proof structures, which require more extensive verification procedures despite their superior proving efficiency.

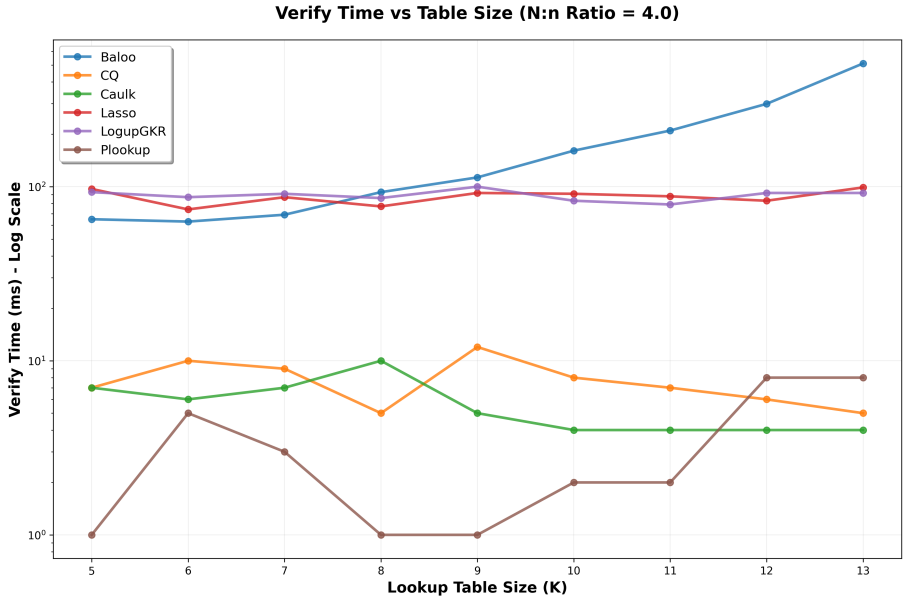


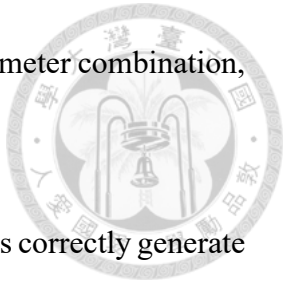
Figure 4.5: Verification time in milliseconds versus lookup table size K for N:n ratio of 4.0. The graph shows the independence of verification time from table size and the performance tier stratification among different protocols.

4.4 Completness and Soundness

To verify the correctness of our lookup argument implementations, we conducted comprehensive completeness and soundness testing using two fundamental operations: addition (add) and range queries. Our testing methodology employed systematic parameter variation across all lookup argument systems, testing with lookup table sizes K ranging

doi:10.6342/NTU202503318

from 6 to 10, and lookup frequency ratios of 2, 4, and 8. For each parameter combination, we executed 10 independent test runs to ensure statistical reliability.



Completeness Testing: We verified that all valid lookup operations correctly generate proofs that pass verification. For completeness validation, we tested scenarios where the lookup arguments should legitimately verify, confirming that our implementations correctly produce valid proofs for authentic lookup queries.

Soundness Testing: We validated the security properties by testing scenarios designed to fail verification. In soundness testing, we deliberately introduced invalid lookup attempts and confirmed that the verification process correctly rejects these malicious or incorrect proofs, thereby ensuring the cryptographic security of each lookup argument system.

4.5 Theoretical and Experimental Analysis

In this section, we conduct a comprehensive analysis by comparing the experimental performance data of each lookup system with their respective theoretical foundations. This comparative analysis provides insights into the practical implications of theoretical complexity and validates the effectiveness of different design approaches.

4.5.1 Plookup

Theoretical Framework: The Plookup protocol exhibits a prover time complexity of $O(N \log N)$, where N represents the table size. Since $N = 2^K$, the system demonstrates high sensitivity to the parameter K. According to theoretical analysis, both proof size

doi:10.6342/NTU202503318

and verification time should remain relatively small and stable across different parameter configurations. The setup time should also correlate linearly with table size N.

Experimental Validation:

Prover Time: The experimental results demonstrate strong alignment with theoretical predictions. Prover time increases significantly with K values, escalating from approximately 30ms at $K = 5$ to over 20 seconds at $K = 13$. This behavior fully conforms to the expected $O(N \log N)$ complexity.

Setup Time: Setup time exhibits a linear relationship with N, growing from approximately 107ms at $K = 5$ to approximately 3.2 seconds at $K = 13$. The performance curves remain virtually unchanged across different $N : n$ ratios, confirming that setup costs are insensitive to lookup density.

Proof Size and Verification Time: Proof size consistently remains at 672 bytes, while verification time maintains stability within the 1-12ms range across all test configurations.

4.5.2 Caulk

Theoretical Framework: The Caulk system presents a preprocessing time complexity of $O(N \log N)$ and a prover time complexity of $O(m^2 + m \log N)$, where m denotes the number of lookups and N represents the table size. This complexity structure implies that increasing K contributes to time growth through the $\log N$ term, while increasing n (equivalent to m) results in quadratic time growth due to the m^2 term.

Experimental Validation:

Setup Time: The data validate the theoretical predictions, with setup time increas-

doi:10.6342/NTU202503318

ing sharply with K, growing from approximately 266ms at $K = 5$ to over 19 minutes (1,154,120ms) at $K = 13$, consistent with the expected $O(N \log N)$ behavior.

Prover Time: Prover time demonstrates extreme sensitivity to both K and ratio parameters. With a fixed ratio of 2, increasing K from 5 to 13 results in prover time escalation from 108ms to nearly 34 minutes (2,040,396ms), clearly demonstrating the dominant influence of the m^2 term. Conversely, with K fixed at 11, varying the ratio from 2 to 16 causes prover time to decrease dramatically from approximately 209 seconds to approximately 1.7 seconds. This behavior perfectly validates the $O(m^2)$ characteristic and confirms Caulk's theoretical bottleneck.

4.5.3 Baloo and CQ

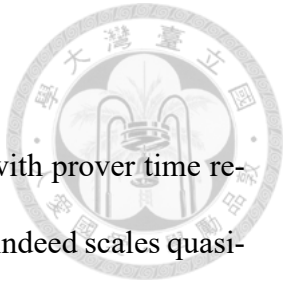
Theoretical Framework: Both Baloo and CQ were designed to address Caulk's quadratic bottleneck, aiming to reduce prover time complexity to near-linear $O(m \log m)$ or $O(m \log^2 m)$. Theoretically, prover time should increase primarily quasi-linearly with n (equivalent to m), while demonstrating reduced sensitivity to ratio variations. Preprocessing time remains dependent on N.

Experimental Validation:

Baloo: The experimental results confirm theoretical predictions. Baloo's prover time is primarily influenced by K, increasing from approximately 75ms at $K = 5$ to over 30 seconds at $K = 13$. Critically, when K remains fixed, prover time demonstrates insensitivity to ratio changes. For instance, at $K = 10$, ratio variations from 2 to 16 result in prover time fluctuations within the narrow range of 984-1025ms, contrasting sharply with Caulk's performance and demonstrating successful mitigation of the $O(m^2)$

doi:10.6342/NTU202503318

bottleneck.



CQ: CQ exhibits performance characteristics similar to Baloo, with prover time remaining insensitive to ratio variations, confirming that its complexity indeed scales quasi-linearly with m. The data support CQ's superior constant factors, particularly for larger K values (e.g., $K = 13$), where CQ's prover time (approximately 4 seconds) significantly outperforms Baloo's (approximately 30 seconds). However, CQ's setup time (approximately 90 seconds at $K = 13$) significantly exceeds Baloo's (approximately 1.8 seconds), reflecting different technical trade-offs.

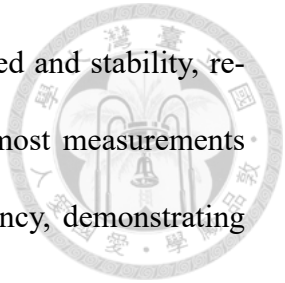
4.5.4 Lasso and LogupGKR

Theoretical Framework: Lasso employs sparse polynomial commitments, making prover costs depend primarily on the number of lookups m rather than the total table size N. LogupGKR utilizes GKR optimization for logarithmic derivative lookups, theoretically reducing the number of polynomials requiring prover commitment, thereby lowering computational costs.

Experimental Validation:

Lasso: The experimental results strongly support theoretical predictions. Lasso's prover time demonstrates exceptional speed and stability, with most tests completing within 30-70ms across all configurations where K ranges from 5 to 13. This performance indicates that prover cost remains virtually independent of table size N, representing a significant practical advantage. Setup time also remains at extremely low levels (approximately 100-130ms), validating the system's low preprocessing dependency.

doi:10.6342/NTU202503318



LogupGKR: The system's prover time exhibits exceptional speed and stability, remaining largely unaffected by both K and ratio parameters, with most measurements falling between 10-116ms. Its performance parallels Lasso's efficiency, demonstrating the effectiveness of the GKR optimization approach.

Overall, Lasso and LogupGKR achieve optimal prover time performance across all evaluated systems, representing the current state-of-the-art in lookup argument efficiency.

4.5.5 Practical Implications and Design Trade-offs

4.5.5.1 Secondary Importance Justification

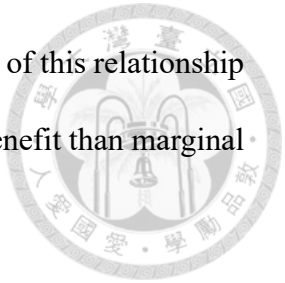
Proof size and verification time represent secondary performance considerations for several fundamental reasons:

Protocol Design Constraints: Modern zero-knowledge proof systems are specifically designed to ensure that both proof sizes and verification times remain practically manageable. The cryptographic foundations of these systems inherently constrain these metrics to acceptable ranges, regardless of the underlying computational complexity.

Proving Time Dominance: In practical deployments, proving time typically represents the primary computational bottleneck. The time required to generate proofs often exceeds verification time by several orders of magnitude, making proving efficiency the critical factor in system performance optimization.

Resource Allocation Considerations: In typical deployment scenarios, proof generation occurs on computationally powerful prover systems, while verification may be

doi:10.6342/NTU202503318



performed by resource-constrained validators. The asymmetric nature of this relationship means that optimizing proving efficiency provides greater practical benefit than marginal improvements in verification performance.

4.5.5.2 Design Philosophy Implications

The observed trade-offs between proving efficiency and verification performance reflect fundamental design philosophy differences:

Prover-Optimized Systems (Lasso, LogupGKR): These protocols prioritize proving efficiency at the expense of increased verification complexity. This design choice is justified in scenarios where proof generation frequency significantly exceeds verification frequency, or where prover resources are more constrained than verifier resources.

Balanced Systems (Plookup, Baloo, CQ): These protocols attempt to achieve reasonable performance across all metrics, accepting moderate proving inefficiency in exchange for superior verification characteristics. This approach is appropriate for applications requiring frequent verification by multiple parties.

4.5.6 Conclusion

The analysis of proof size and verification time provides valuable insights into the comprehensive performance characteristics of lookup argument systems. While these metrics represent secondary considerations compared to proving time and setup efficiency, they reveal important design trade-offs that influence protocol selection for specific deployment scenarios. The fundamental independence of verification time from computa-

doi:10.6342/NTU202503318

tional complexity validates the theoretical foundations of modern SNARK systems and confirms their suitability for practical zero-knowledge applications.

doi:10.6342/NTU202503318

Chapter 5 Conclusion and Future Work

This thesis embarked on a comprehensive investigation into the practical performance of modern zero-knowledge lookup arguments, a cornerstone technology for enhancing the scalability of blockchain systems like ZK rollups. Motivated by the understanding that theoretical asymptotic complexity does not solely determine real-world efficiency, our primary objective was to bridge the gap between theory and practice through rigorous, empirical benchmarking. By implementing and systematically evaluating a suite of prominent lookup protocols—from the foundational Plookup to the state-of-the-art Lasso and LogupGKR—within a unified framework, we have generated concrete data to guide developers and researchers in this rapidly advancing field.

5.1 Summary of Key Findings

Our experimental evaluation successfully charted the performance evolution of lookup arguments, providing empirical validation for the field's theoretical advancements. The key findings are summarized as follows:

- Validation of the Evolutionary Path: We empirically confirmed the performance narrative of lookup arguments. Plookup's prover time demonstrated a clear linear

doi:10.6342/NTU202503318

dependency on table size ($O(N)$), making it suitable for small tables but impractical for large-scale applications. We then validated the critical performance bottleneck of Caulk, whose $O(n^2)$ complexity in the number of lookups (n) severely limits its utility, despite being the first to achieve sublinearity in table size. Its successors, Baloo and CQ, were shown to effectively resolve this bottleneck, exhibiting quasi-linear performance ($O(n \log n)$) that remains stable even with a high volume of lookups.

- Paradigm Shift in Prover Performance: The most significant finding of this study is the paradigm-shifting performance of Lasso and LogupGKR. These protocols delivered prover times that were consistently orders of magnitude faster than their predecessors. Their performance exhibited remarkable stability across varying table sizes and lookup counts, underscoring the profound impact of novel techniques such as sparse polynomial commitments (Lasso) and GKR-based optimizations for logarithmic derivatives (LogupGKR).

- Elucidation of Practical Trade-offs: A central conclusion drawn from our analysis is that there is no universally superior lookup protocol. The selection is a nuanced engineering decision dictated by application-specific requirements. We quantified a multi-dimensional trade-off involving:

  - Prover Time vs. Proof Size: Lasso and LogupGKR offer unparalleled prover speed at the cost of larger proof sizes compared to pairing-based schemes.

  - Preprocessing vs. Prover Time: Protocols like CQ and Caulk require significant, table-dependent preprocessing, making them suitable for applications with large, static tables where this one-time cost can be amortized. In con-

doi:10.6342/NTU202503318

trast, the low-to-zero preprocessing overhead of Lasso and LogupGKR makes them ideal for dynamic or extremely large structured tables.

– Verification Time and Aggregation: Pairing-based schemes like Plookup and CQ offer extremely fast verification times and, in CQ's case, native aggregatability, which is a critical feature for recursive proof systems.

5.2 Limitations of the Study

While this study provides a comprehensive benchmark, it is subject to certain limitations that offer context for the results:

- Implementation-Specific Performance: The performance was evaluated on a single, albeit unified, Rust implementation based on the 'plonkish' backend. Results could differ with other cryptographic libraries, programming languages, or low-level optimizations not explored in this work.
- Hardware Dependency: All benchmarks were executed on a specific hardware configuration. Performance on different architectures, particularly those with varying core counts and cache hierarchies, may differ, especially for highly parallelizable protocols.
- Scope of Parameters: The tested range of table sizes ($N = 2^5$ to 2^{11}) and lookup ratios, while broad, does not cover all possible scenarios. The performance advantages of sublinear-N protocols might become even more pronounced at extremely large N/n ratios not tested here.

doi:10.6342/NTU202503318

- Standalone Primitives: The protocols were benchmarked as standalone cryptographic primitives. They were not integrated into a fully-fledged, end-to-end ZK-rollup or ZK-VM system, which would introduce additional overheads from the constraint system, front-end compiler, and on-chain components.

5.3 Future Work and Open Questions

The benchmarking framework presented in this paper provides a foundational comparison of contemporary lookup arguments. However, several dimensions remain to be explored to fully understand their practical trade-offs and guide future research. This section outlines key areas for future investigation.

5.3.1 Expanding Benchmarking Scenarios

Our current evaluation focuses on single-column lookups into static tables. Real-world applications often present more complex requirements.

5.3.1.1 Dynamic and Vector Lookups

Future work should extend the benchmark suite to include:

- Dynamic Tables: Scenarios where the table's content is generated by the prover during proof execution, such as in modeling RAM state. This would critically test the performance of protocols in environments where extensive preprocessing is infeasible. The online computational cost for Plookup and LogUpGKR, and the ap-

doi:10.6342/NTU202503318

plicability of preprocessing for Baloo and CQ, would be key areas of analysis.

- Vector Lookups: Practical applications frequently require lookups of tuples or multi-column records (e.g., `(address, value, timestamp)`). An essential extension is to implement and evaluate strategies for handling such lookups, particularly the overhead associated with techniques like random linear combinations (RLC). This analysis should quantify the additional computational burden on the prover and any resulting increase in proof size.

5.3.1.2 Performance in Recursive and Accumulative Settings

Certain protocols, such as those employed in Lasso and Proofs for Deep Thought, are explicitly optimized for recursive proof composition. A valuable line of inquiry would be to design a benchmark that simulates a recursive or incremental computation setting, where each step performs a limited number of lookups. Such a test would measure the cost of the accumulation prover and could highlight the strengths of CQ, with its aggregation-friendly verifier, and LogUpGKR, due to its potentially low single-step overhead.

5.3.2 Analysis of Advanced Protocol Features

Beyond raw performance, the structural properties of these protocols have significant practical implications.

doi:10.6342/NTU202503318

5.3.2.1 Homomorphism and Aggregatability

- Homomorphic Properties: The support for homomorphic commitments in Baloo and CQ (via KZG) is a powerful feature, particularly for vector lookups. A targeted benchmark could be designed to quantify this benefit, for instance, by comparing the cost of a single batched proof for a vector lookup against proving multiple independent lookups.

- Proof Aggregation: A more detailed analysis is needed regarding proof composability. We should elaborate on how CQ's use of a fixed-base G2 element in its verifier facilitates straightforward integration with recursive SNARKs (e.g., Nova-style accumulation). In contrast, the challenges posed by Baloo's variable-base G2 point, $[z_I]_2$, which complicates standard aggregation techniques, should be thoroughly investigated.

5.3.2.2 Cross-Implementation Benchmarking

To distinguish between protocol-inherent characteristics and implementation-specific artifacts, it would be beneficial to compare the same protocol across different backend cryptographic libraries (e.g., `arkworks` versus the original `halo2` codebase). This would help isolate performance bottlenecks and provide a more normalized comparison.

5.3.3 Application-Oriented Protocol Selection

Ultimately, the choice of a lookup argument is application-dependent. Future analysis should focus on creating a clear mapping between application profiles and protocol

doi:10.6342/NTU202503318

strengths.

- Large, Static Tables ($N \gg n$): For use cases like range checks or cryptographic primitive lookups (e.g., AES S-boxes), CQ and an optimized Baloo appear to be strong candidates, provided the one-time preprocessing cost is amortizable.

- Dynamic Tables ($N \approx n$): In scenarios like RAM or state machine modeling, the low setup cost of Plookup and LogUpGKR may offer a decisive advantage where the preprocessing of Baloo or CQ is inapplicable.

- Recursive Applications: For lookup-intensive recursive computations (e.g., Jolt or the ”lookup singularity”), the primary candidates are protocols designed for this paradigm, including CQ (due to aggregatability) and LogUpGKR (due to low single-step overhead).

doi:10.6342/NTU202503318

References

[1] A. Gabizon and D. Khovratovich, "Plookup: A simplified polynomial protocol for lookup tables," Cryptology ePrint Archive, 2020.

[2] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Caulk: Lookup arguments in sublinear time," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 3121–3134, 2022.

[3] A. Zapico, A. Gabizon, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Baloo: Nearly optimal lookup arguments," Cryptology ePrint Archive, 2022.

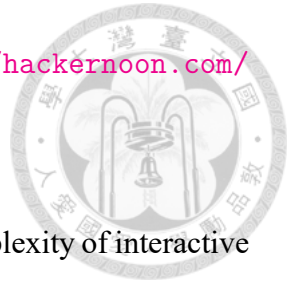
[4] L. Eagen, D. Fiore, and A. Gabizon, "cq: Cached quotients for fast lookups," Cryptology ePrint Archive, 2022.

[5] S. Papini and U. Haböck, "Logup-gkr: A more efficient approach for proving lookups," Cryptology ePrint Archive, 2023.

[6] S. Setty, J. Thaler, and R. Wahby, "Lasso: lookup arguments for rlc-based snarks," Cryptology ePrint Archive, 2023.

[7] U. B. RDI, "Zk learning group lecture 12: Zkvm and zkevm." https://rdi.berkeley.edu/zk-learning/assets/lecture12.pdf. Accessed: 2024.

doi:10.6342/NTU202503318



[8] HackerNoon, "Exploring lookup arguments." https://hackernoon.com/exploring-lookup-arguments. Accessed: 2024.

[9] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," in SIAM Journal on computing, vol. 18, pp. 186–208, SIAM, 1989.

[10] V. Buterin, "An incomplete guide to rollups." https://vitalik.ca/general/2021/01/05/rollup.html, 2021.

[11] P. Team, "zkevm: Scaling ethereum with zero knowledge proofs," Technical Report, 2022.

[12] R. Z. Team, "Risc zero: A zero-knowledge virtual machine," Technical Report, 2022.

[13] S. Bowe, J. Grigg, and D. Hopwood, "Halo 2," Cryptology ePrint Archive, 2019.

[14] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," in Proceedings of the seventeenth annual ACM symposium on Theory of computing, pp. 291–304, 1985.

[15] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 326–349, 2012.

[16] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in Annual international conference on the theory and applications of cryptographic techniques, pp. 626–645, Springer, 2013.

doi:10.6342/NTU202503318

[17] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali, and P. Rogaway, "Everything provable is provable in zero-knowledge," in Conference on the Theory and Application of Cryptography, pp. 37–56, Springer, 1988.

[18] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in International conference on the theory and application of cryptology and information security, pp. 177–194, Springer, 2010.

[19] B. Parno, M. Raykova, and V. Vaikuntanathan, "Succinct arguments from multi-prover interactive proofs and their efficiency benefits," in Annual Cryptology Conference, pp. 255–272, Springer, 2013.

[20] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in Annual International Cryptology Conference, pp. 733–764, Springer, 2019.

[21] D. Catalano and D. Fiore, "Vector commitments and their applications," in Public-Key Cryptography–PKC 2013, pp. 55–72, Springer, 2013.

[22] R. W. Lai and G. Malavolta, "Subvector commitments with application to succinct arguments," Cryptology ePrint Archive, 2019.

[23] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in 2018 IEEE symposium on security and privacy (SP), pp. 315–334, IEEE, 2018.

[24] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in 2018 IEEE symposium on security and privacy (SP), pp. 315–334, IEEE, 2018.

doi:10.6342/NTU202503318

[25] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," Journal of the ACM, vol. 27, no. 4, pp. 701–717, 1980.

[26] R. Zippel, "Probabilistic algorithms for sparse polynomials," in International symposium on symbolic and algebraic manipulation, pp. 216–226, 1979.

[27] A. Fiat and A. Shamir, "How to prove yourself: practical solutions to identification and signature problems," in Conference on the Theory and Application of Cryptographic Techniques, pp. 186–194, Springer, 1986.

[28] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in Theory of Cryptography Conference, pp. 31–60, Springer, 2016.

[29] N. Ron-Zewi and R. D. Rothblum, "Fast reed-solomon interactive oracle proofs of proximity," in International Colloquium on Automata, Languages, and Programming, pp. 1–14, 2016.

[30] J. Teutsch and C. Reitwiessner, "Truebit: A scalable verification solution for blockchains," arXiv preprint arXiv:1908.04756, 2017.

[31] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in 27th USENIX Security Symposium, pp. 1353–1370, 2018.

[32] P. Team, "Polygon zkevm: A zk-rollup with ethereum virtual machine opcodes." https://polygon.technology/polygon-zkevm, 2022.

[33] M. Labs, "Matter labs zksync 2.0: A zk-rollup using zero-knowledge proofs." https://docs.zksync.io/, 2022.

[34] S. Team, "Scroll: Native zkevm layer 2 for ethereum." https://scroll.io/, 2023.

doi:10.6342/NTU202503318

[35] R. Zero, "Risc zero: A zero-knowledge virtual machine." https://risczero.com/, 2023.

[36] S. Labs, "Sp1: A performant, open-source zero-knowledge virtual machine." https://github.com/succinctlabs/sp1, 2024.

[37] J. Thaler et al., "Jolt: Snarks for virtual machines via lookups." https://jolt.a16zcrypto.com/, 2024.

[38] Iden3, "Circom: A circuit compiler for zero knowledge proofs." https://docs.circom.io/.

[39] J. Eberhardt et al., "Zokrates: A toolbox for zksnarks on ethereum." https://zokrates.github.io/.

[40] Aleo, "Leo: A functional, statically-typed programming language built for writing private applications." https://leo-lang.org/.

[41] M. Labs, "Zinc: A framework for zk-snark development." https://zinc.matterlabs.dev/.

[42] StarkWare, "Cairo: A turing-complete stark-friendly cpu architecture." https://www.cairo-lang.org/.

[43] A. Protocol, "Noir: A domain specific language for snark proving systems." https://noir-lang.org/.

[44] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," in Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 517–552, Springer, 2019.

doi:10.6342/NTU202503318

[45] NIST, "Secure hash standard (shs)," Tech. Rep. FIPS PUB 180-4, National Institute of Standards and Technology, 2015.

[46] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," in Proceedings of the fortieth annual ACM symposium on Theory of computing, pp. 113–122, 2008.

[47] L. Pearson, J. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia, "Plonkup: Reconciling plonk with plookup," Cryptology ePrint Archive, 2022.

[48] U. Haböck, "Multivariate lookups based on logarithmic derivatives," Cryptology ePrint Archive, 2022.

[49] J. Ernstberger, S. Chaliasos, G. Kadianakis, S. Steinhorst, P. Jovanovic, A. Gervais, B. Livshits, and M. Orrù, "zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks," 2023. Technical University of Munich, Imperial College London, Ethereum Foundation, University College London, Centre National de la Recherche Scientifique.

[50] N. Gailly et al., "zk-benchmarking: Benchmarking zk-circuits in circom." https://github.com/delendum-xyz/zk-benchmarking, 2023. Delendum Research.

[51] M. El-Hajj et al., "Evaluating the efficiency of zk-snark, zk-stark, and bulletproof in real-world scenarios: A benchmark study," Information, 2024. Systematic Review, 3 citations.

[52] C. Steidtmann et al., "Benchmarking zk-circuits in circom," IACR Cryptology ePrint Archive, 2023. 2 citations.

doi:10.6342/NTU202503318

[53] H. Guo et al., "Benchmarking zk-friendly hash functions and snark proving systems for evm-compatible blockchains," arXiv preprint, 2024. 1 citation.

[54] han0110, "plonkish: A zksnark building block framework." https://github.com/han0110/plonkish/commit/303cf244803ea56d1ac8c24829ec4c67e4e798ab, 2023. Accessed: 2024-07-28.

[55] han0110, "plonkish_backend in plonkish framework." https://github.com/han0110/plonkish/tree/main/plonkish_backend, 2023. Accessed: 2024-07-28.

[56] Z. Foundation, "halo2: A zk-snark library." https://github.com/zcash/halo2. Accessed: 2024-07-28.

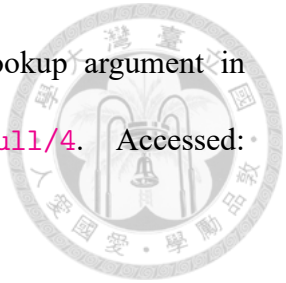
[57] caulk crypto, "caulk: An implementation of the caulk lookup argument." https://github.com/caulk-crypto/caulk/commit/8210b51fb8a9eef4335505d1695c44ddc7bf8170. Accessed: 2024-07-28.

[58] geometryxyz, "cq: An implementation of the cq lookup argument." https://github.com/geometryxyz/cq/commit/c0e499cdf866631b5079a2ae6837e26df784d0eb. Accessed: 2024-07-28.

[59] han0110, "Fractional sum check implementation in plonkish." https://github.com/han0110/plonkish/blob/main/plonkish_backend/src/piop/gkr/fractional_sum_check.rs. Accessed: 2024-07-28.

[60] a16z crypto, "Jolt: Snarks for virtual machines via lookups." https://github.com/a16z/jolt/commit/a2eb0ad5bc1b96b73480b2dc4d95199e2efe3a7a. Accessed: 2024-07-28.

doi:10.6342/NTU202503318



[61] DoHoonKim8, "halo2-lasso: An implementation of lasso lookup argument in halo2." https://github.com/DoHoonKim8/halo2-lasso/pull/4. Accessed: 2024-07-28.

[62] nooma 42, "Lasso integration modifications." https://github.com/nooma-42/Lookup-Argument/commit/47acf4f764586fc3e83cea54de60e002c477b6b2. Accessed: 2024-07-28.

doi:10.6342/NTU202503318