Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2023.0322000

ZCLS: A Lifecycle Strategy for Efficient ZK-Rollup Circuit Optimization in Circom

KHOA TAN VO[1,2,*], MINH NGO[1,2,*], THU NGUYEN[1,2], THU-THUY TA[1,2], MONG-THY NGUYEN THI[1,2], HONG-TRI NGUYEN[3], TU-ANH NGUYEN-HOANG[1,2]

[1]Faculty of Information Science and Engineering, University of Information Technology, Ho Chi Minh City, Vietnam
[2]Vietnam National University, Ho Chi Minh City, Vietnam
[3]Aalto University, Finland
[*]These authors contributed equally

Corresponding author: Khoa Tan VO (e-mail: khoavt@uit.edu.vn, n240203@grad.uit.edu.vn) and Hong-Tri Nguyen (e-mail: hong-tri.nguyen@aalto.fi).

This research was supported by The VNUHCM-University of Information Technology's Scientific Research Support Fund.

ABSTRACT Scalability remains a key challenge for layer 1 blockchains. ZK-Rollups, leveraging zero-knowledge proofs, offer a promising layer 2 solution by improving throughput and reducing costs while preserving security. However, the performance of ZK-Rollup still poses a major barrier to practical implementation. The proving circuits in popular applications like ERC-20 transactions are highly complex, often containing a large number of constraints, which directly impacts the computation time and resources required to generate zero-knowledge proofs. This study presents an empirical study on the impact of constraint optimization in Circom on the performance of ERC-20 ZK-Rollups using Groth16. Three optimization levels (–O0, –O1, –O2) are evaluated across transaction batches of 4 to 64. Results show a trade-off: –O2 reduces constraints by up to 73.2% but increases compilation time by 162.75% at batch size 64, while –O1 offers a more balanced approach suitable for development stages. Findings confirm that proof generation time is closely tied to constraint count and complexity. Based on these insights, this study introduces ZCLS (ZK-Circuit Lifecycle Strategy), a practical framework for selecting optimization flags aligned with development stages to enhance ZK-Rollup system efficiency.

INDEX TERMS constraint optimization, experimental analysis, groth16, zero- knowledge proofs, zk-rollup, zk-snarks

I. INTRODUCTION

As blockchain technology continues to evolve, the need for scalability and improved transaction throughput has become increasingly urgent [1] [2] [3]. Among various solutions, such as sharding, side-chains, and state channels, Zero-Knowledge (ZK)-Rollup has emerged as a prominent layer 2 approach, widely deployed in systems like ZKSync Era, Starknet, and Polygon zkEVM [4]. By aggregating off-chain transactions into batches and generating succinct Zero-Knowledge Proofs (ZKP) [5], ZK-Rollups significantly reduce on-chain data and computation, improving throughput and lowering gas fees while maintaining security guarantees from the underlying layer 1 blockchain [6].

Despite its promise, practical implementation of ZK-Rollups faces persistent challenges, particularly stemming from the complexity of ZK proving circuits [7]. Manual circuit development is error-prone [8], and the constraint count directly impacts compilation time, proving time, and resource requirements [9]. While new proving systems such as STARK [10], Bulletproof [11], and Plonk [12] have been proposed [13], a critical gap remains at the circuit and compiler level—specifically in terms of performance optimization, where developers encounter optimization challenges first [14].

This study addresses that gap by investigating the impact of constraint optimization in Circom [9] [15], a widely used tool for zk-SNARK circuit development. Circom provides multiple optimization levels (–O0, –O1, –O2), which represent increasing levels of constraint simplification applied during circuit compilation. Understanding the effects of these optimization flags on constraint complexity provides valuable insights for developers during the circuit design phase, prior to engaging with system-level optimizations.

To bridge the gap between developer-level circuit design and system-level performance tuning, we introduce the ZK-Circuit Lifecycle Strategy (ZCLS) — a practical guideline for selecting Circom optimization flags based on empirical obser-

vations of their trade-offs at various stages of development and deployment. In parallel, we also introduce CirMetrics, a supporting tool aimed at helping developers monitor circuit metrics in real time and apply ZCLS recommendations more effectively. Accordingly, this study poses the following research questions:

- RQ1: How do the levels of constraint optimization affect complex ZK circuits, such as the verification circuits for ZK-Rollups?
- RQ2: Does the proposed ZCLS method contribute to performance improvements for developers when utilizing Circom's optimization flags during the development of ZK circuits?
- RQ3: What is the scalability and adaptability of ZCLS in relation to practical systems?

To address these questions, the study defines the following objectives:

1) Conduct a comprehensive survey of optimization techniques in zk-SNARK proving systems to identify research gaps (e.g., [10]–[13], [16]–[19]).
2) Perform quantitative analysis of optimization flags on constraint count, compilation time, proving time, peak RAM usage measured during the compilation and proving phases, and on-chain verification costs.
3) Analyze trade-offs between optimization level, transaction batch sizes, and system performance.
4) Investigate the impact of –O2 optimization on the balance between linear and non-linear constraints.
5) Propose a practical guideline, ZCLS, for selecting Circom optimization flags based on empirical evidence, tailored to development stages and deployment requirements.

By clarifying the role of constraint optimization in improving ZK-Rollup proving efficiency, this study contributes toward minimizing proving costs, accelerating development cycles, and facilitating the integration of ZKPs into scalable blockchain applications.

The remainder of this paper is organized as follows. Section II reviews previous work on ZKP optimization. Section III introduces key theoretical concepts including blockchain, ZK-Rollups, zk-SNARKs, Groth16, and Circom compiler optimization. Section IV outlines our experimental setup, system architecture, and the proposed ZCLS. Section V presents and discusses the experimental results and evaluation. Section VI concludes the paper and suggests future work.

II. RELATED WORK

Recent research on optimizing zero-knowledge proof generation performance, particularly concerning zk-SNARKs, can be broadly categorized into two main directions: back-end optimization (proving system) and front-end optimization (constraint/circuit system for proof generation) [17] as illustrated in Fig. 1. The figure provides a high-level overview of the architecture and interactions among key components in zk-SNARK-based systems. A more detailed explanation of each layer and transformation pipeline from circuit definition to final proof is presented in III-A1.

A. BACK-END OPTIMIZATION

Efforts in this area focus on proposing new and more efficient ZKP schemes, such as GENES [20] – a new recursive zk-SNARK that does not require trusted setup, designed to address issues of proof generation and verification time in ZKP applications, especially on blockchains. GENES combines recursive proofs, merging multiple instances of a Rank 1 Constraint System (R1CS) [21] into one, and distributes computation by using ''helpers'' to disperse proof commitments. This ZKP scheme stands out with its constant O(1) verification time, despite the drawback of larger proof sizes compared to other schemes.

HyperPlonk [22] is a new proving system developed based on Plonk, using a special data structure called a boolean hypercube and applying a polynomial commitment technique called multilinear polynomial commitment to enhance efficiency. Compared to traditional Plonk, HyperPlonk offers several benefits: it eliminates the need for complex FFT (Fast Fourier Transform) computations and handles custom "gates" (operations) with higher complexity more efficiently. These improvements significantly reduce the time required to generate proofs. Additionally, much research has aimed to enhance existing ZKP schemes, such as Groth16, with variants like Polymath [23] – a zk-SNARK designed for Square Arithmetic Programming (SAP) constraint systems instead of Groth16's R1CS, using KZG (Kate-Zaverucha-Goldberg Polynomial Commitments) polynomial commitment schemes. Its main goal is to achieve significantly shorter proof sizes, especially at higher security levels. The overall objective of these methods is to reduce proof generation time, proof size, and verification time, or to eliminate the need for trusted setup.

B. FRONT-END OPTIMIZATION

1) Optimization for ZK circuit security

Research related to optimizing constraints for ZK circuits has also attracted much attention. However, most of these works primarily focus on security aspects, addressing issues related to circuit correctness. This includes identifying and fixing problems related to under-constrained circuits (circuits that accept invalid proofs, leading to serious security vulnerabilities, such as exploits in zkSync allowing asset theft [24]) and over-constrained circuits (circuits that reject valid proofs, affecting the completeness of the system). Notable studies addressing these issues include AC^4 [25]—a tool for modeling Circom circuits as polynomial systems and applying algebraic algorithms to check for over- or under-constrained issues; QED^2 (Picus) [18]—an automated technique and tool designed to detect under-constrained ZKP circuits by combining lightweight unique constraint propagation (UCP) with Satisfiability Modulo Theories (SMT) solvers; Circomspect [26]—a static analysis tool (examining the source code of circuits without executing the code) to analyze and identify

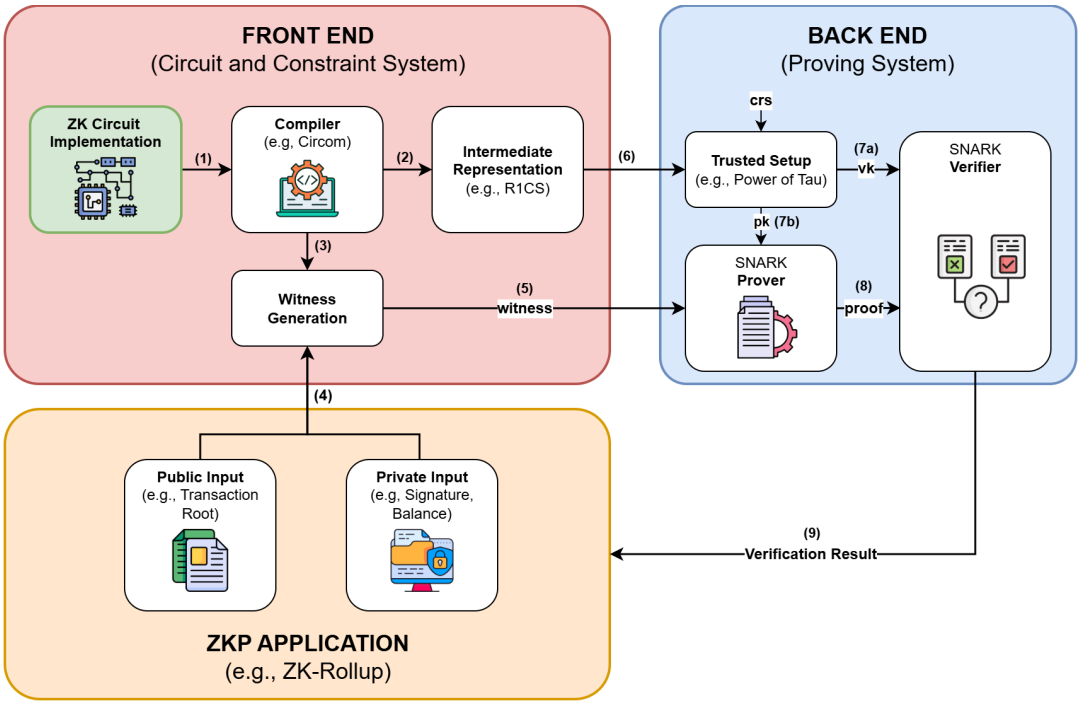

FIGURE 1. **The relationship between the front-end and back-end of zk-SNARKs systems and the applications utilizing this system. Regarding the sequence numbers along with arrows, (1)–(9) denote the sequential flow of circuit compilation, witness generation, proof construction, and verification. (6*) indicates the one-time Trusted Setup phase, where the common reference string (crs) is used to generate the proving key (pk) and verification key (vk). (7a) and (7b) refer to the use of pk and vk by the Prover and Verifier, respectively.**

security vulnerabilities in ZKP circuits; and Veridise's Coda [27]—a tool that uses formal verification to check the properties and correctness of ZKP circuits.

2) Optimization for constraint simplification in ZK circuits

In terms of optimizing constraint systems to enhance ZKP generation performance, these works focus on streamlining and minimizing the number of constraints in ZK circuits to improve proof generation capability. For example, the work by Albert et al. [19], focuses on simplifying constraints from R1CS by removing parts of constraints that do not affect security. Specifically, they aim to identify and eliminate redundant constraints by using a framework for R1CS reduction based on a rule-based transformation system and a new constraint optimization technique based on Gaussian elimination to infer linear constraints from non-linear constraints, with the goal of reducing redundant constraints in the R1CS circuit, thereby reducing the cost of zero-knowledge proof generation. This research also states that it is the first work to study techniques applied to constraints as a formal method to address challenges in analyzing and optimizing ZK protocols.

It can be seen that, despite the existence of research on front-end optimization, these studies often focus on circuit correctness (security). A significant gap remains in research to simplify constraints similar to the work of Albert et al. [19], aimed at increasing ZKP generation performance, and also lacks quantitative empirical analysis of the impact of compiler optimization options (such as flags in Circom) on performance metrics (compilation time, proof generation time and peak RAM usage for the two tasks described above) in the context of complex ZK-Rollup ERC-20 applications.

This research aims to fill this gap by quantifying performance trade-offs and clarifying the –O2 optimization mechanism, focusing on reducing the total number of constraints in core cryptographic operations of ZK-Rollup for ERC-20 [28] transactions. From there, this study proposes a circuit optimization framework, supported by empirical data, to provide practical guidance for improving ZK-Rollup performance through effective constraint management. Table 1 summarizes key related works and situates our contribution in optimizing ZK circuits for performance.

III. BACKGROUND KNOWLEDGE

This section will provide the theoretical foundation for the topic with the following contents:

- zk-SNARKs and Groth16
- Circom compiler and constraint optimization levels
- Analysis of –O2 optimization level

A. ZK-SNARKS AND GROTH16

1) zk-SNARKs

zk-SNARK [29] stands for "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge." It is a type of zero-knowledge proof that allows one party (the prover) to convince another party (the verifier) that they know a solution (witness) to a specific computational problem without re-

TABLE 1. Summary of studies on ZKP Schemes and Circuit Optimization

Study	Summary	Improvements to ZKP Schemes	Security Enhancements for ZK Circuits	Optimizations for ZK Constraints	Performance Trade-offs in Circuit Optimization	Proposed Circuit Optimization Framework
HyperPlonk [22] (2023)	Uses boolean hypercube, multilinear commitment, eliminates FFT, handles complex gates.	✓				
Polymath [23] (2024)	zk-SNARK SAP, KZG, reduces proof size at high security levels.	✓				
GENES [20] (2025)	Recursive zk-SNARK, no trusted setup, O(1) verification, larger proofs.	✓				
Circomspect [26] (2022)	Identifies security vulnerabilities through Circom circuit analysis.		✓			
QED^2 (Picus) [18] (2023)	Detects under-constrained issues automatically using UCP + SMT.		✓			
Veridise's Coda [27] (2024)	Verifies the correctness properties of ZKP circuits.		✓			
AC^4 [25] (2024)	Detects under/over-constrained issues via polynomial modeling and algebraic methods.		✓			
Albert et al. [19] (2022)	Optimizes R1CS through rule-based transformations and Gaussian elimination.			✓		
Our Study	Framework for selecting optimization levels in Circom circuits.			✓	✓	✓

vealing any information about the solution itself beyond its existence.

The structure of a zk-SNARK system in practice, explaining Fig. 1. The first component is the ZK circuit layer, where developers write circuits that represent logic using SNARK-friendly languages such as DSLs (Domain-Specific Languages) like Circom, or eDSLs (Embedded Domain-Specific Languages) like Halo2. Circuits have two tasks: first, to compute values and generate witnesses, and second, to check the correctness of the witnesses. The front-end layer is typically represented as a compiler, with circuits as input, and constraints in an arithmetic representation, such as R1CS, as output. The front-end also provides witness generation functionality, which takes public and private values, combines them with the written circuit, and generates the corresponding witness. The back-end layer is typically represented as a proving tool like Snarkjs. It supports functions including: trusted setup, ZKP generation, and ZKP verification. Lastly, the ZKP application layer, along with circuits and the front-end and back-end for ZKP generation and proving, requires additional supporting components to interact with the SNARK elements mentioned above. For example, smart contracts in blockchain technology include verification functions that validate submitted ZKPs and execute corresponding logic based on the verification results.

2) Groth16

Groth16 is one of the most widely used zk-SNARK protocols today, proposed by Jens Groth in 2016 [30]. It is widely adopted due to several significant advantages. First, Groth16 produces extremely succinct proofs, consisting of only three elements on an elliptic curve [31], regardless of the circuit size. Second, it offers fast verification time, requiring only a few pairing operations on the elliptic curve, which makes it highly suitable for deployment in resource-constrained environments such as blockchains. Finally, Groth16 is non-interactive, requiring only a single message from the prover to the verifier, which reduces communication overhead and improves efficiency. However, Groth16 requires a trusted setup phase for each circuit, and if compromised, it can affect the security of the system. Nevertheless, solutions like multi-party computation have been implemented to mitigate this risk in practice.

This study chose Groth16 for ERC-20 transactions on ZK-Rollup because optimizing verification time and proof size is crucial to accommodate a large number of transactions along with continuous proof verification requirements. Groth16 provides very fast proof verification time, regardless of the circuit complexity or batch size. Furthermore, it provides very small proofs; Groth16 is one of the systems with the smallest proof sizes available [32]. Moreover, Circom also provides good support for zk-SNARK proving systems like Groth16, facilitating circuit optimization and extraction of necessary parameters for convenient evaluation.

3) Proof Generation and Verification Flow

The proof generation and verification flow [33] [17] can be summarized as follows, providing a complement to the overall system architecture shown in Fig. 1:

Circuit (R1CS) $\rightarrow$ QAP $\rightarrow$ Trusted Setup $\rightarrow$ Groth16 Proving $\rightarrow$ Verifier

The process begins by representing the arithmetic circuit of an operation. This arithmetic circuit is constructed using addition and multiplication gates, with wires carrying values belonging to a finite prime field F_p. The circuit receives input signals to perform addition and multiplication modulo a prime number p. The output of each operation generates a

signal, which can be an intermediate signal or the final output signal of the computation process. High-level circuit programming languages like Circom are often used to describe these circuits. A circuit will be defined by public signals, private signals, and the R1CS constraint system will describe the circuit's operations on these signals.

To convert the arithmetic circuit into an R1CS system, R1CS is a common intermediate representation for ZKP circuits, used by front-end tools like Circom and ZoKrates [34]. The R1CS representation consists of a set of constraints, each having a rank-one constraint corresponding to a multiplication gate in the circuit, expressed as $a \times b = c$. The set of all such constraints forms a rank-one constraint system. To be compatible with zk-SNARK cryptographic techniques, the R1CS system is further converted into a "Quadratic Arithmetic Program" (QAP). QAP encodes the R1CS constraints into algebraic polynomials, where witness values are embedded into the coefficients of these polynomials. If a valid witness exists, the polynomial $Q(x)$ (constructed from the witness and constraints) will be divisible by a target polynomial $Z(x)$ (defined by the circuit structure). Therefore, the prover must prove the existence of a polynomial $H(x)$ such that $Q(x) = H(x) \times Z(x)$, without directly revealing the witness or these polynomials. The conversion to QAP is a crucial step for applying efficient and secure zero-knowledge proving protocols.

The trusted setup phase is a crucial step to generate the cryptographic keys necessary for proof generation and verification. For Groth16, this process typically involves two phases. Phase 1 (Powers of Tau – a multi-party computation protocol) generates common random secret values for the entire system. Phase 2 (circuit-specific setup) uses the results from Phase 1 and the specific circuit structure (via QAP) to generate the proving key and verification key. Ensuring the secrecy of the random values generated during this process is critical to maintaining the security of the system.

In the proving process, the prover uses the proving key (generated during the trusted setup phase) and the witness (secret values satisfying the circuit) to generate a proof π. Public input values are also provided for the verifier to use during the verification process. The proof π is typically very small. Finally, the verifier uses the verification key (also generated during the trusted setup phase), the proof π received from the prover, and the public input values to check the validity of the proof. The verification process typically uses a few pairing operations on the elliptic curve, and the result will be either acceptance or rejection of the proof.

B. CIRCOM COMPILER AND CONSTRAINT OPTIMIZATION LEVELS

1) Circom Compiler

Circom [9] [15] is a Circuit Description Language and a compiler specifically designed to build arithmetic circuits for ZKPs, especially zk-SNARKs. Circom was developed to simplify the process of designing and implementing complex ZKP circuits, which often require deep knowledge of cryptography and mathematics. It allows developers to define computations as arithmetic circuits more intuitively, then compile them into the R1CS format that zk-SNARK systems can use.

One notable feature of Circom is its programming style, which resembles hardware description languages. This style allows programmers to explicitly define each constraint, providing maximum control over the generated constraints and facilitating the creation of compact, efficient, and cost-effective ZKPs. Circom also supports modularity through the use of templates. Templates allow the creation of reusable and parameterizable sub-circuits. This helps build larger circuits from basic components, enhancing code reusability and reducing complexity during circuit implementation.

Regarding the compilation target, the primary function of the Circom compiler is to transform high-level circuit definitions into R1CS constraints. In addition to the constraint system, the compiler also generates auxiliary files, such as WebAssembly (WASM) or C++ code, to support witness computation. Circom benefits from a growing ecosystem that includes circomlib (a library of common circuit templates) and snarkjs (a JavaScript library for generating and verifying zk-SNARK proofs), along with other supporting tools like circomspect, circomkit, etc., making it easier for developers to build ZK applications.

2) Circom's Constraint Optimization Mechanism

In zk-SNARK systems, the number of constraints in the R1CS circuit has a direct impact on the efficiency of the proving process, particularly in terms of proving time. A reduction in the total number of constraints typically leads to faster proof generation. To facilitate constraint minimization, Circom offers multiple optimization flags that apply different levels of algebraic simplification during circuit compilation. These optimizations aim to eliminate redundant constraints and simplify arithmetic expressions while preserving correctness.

--O0 (No optimization) is the lowest optimization level, where the compiler does not apply any constraint optimization techniques. Each operation in the Circom circuit will be directly converted into corresponding R1CS constraints without any simplification.

--O1 (Basic optimization - Default) is the default level applied by the Circom compiler. It applies basic but effective optimizations to reduce the number of constraints. Key optimizations at the –O1 level include eliminating constraints of the form "signal = k" (where k is a constant) and "signal1 = signal2". The compiler replaces all occurrences of "signal" with k or replaces "signal1" with "signal2" (or vice versa, depending on whether one of the signals is private to maintain proof correctness).

--O2 (Advanced optimization) includes all –O1 optimizations while applying additional advanced techniques, particularly through a "lazy" form of Gaussian Elimination. The core principle of –O2 optimization is based on the observation that addition gates in arithmetic circuits are considered "free" in zk-SNARK systems like Groth16 [19] [35], meaning that the introduction of addition operations does not increase the

number of constraints. To leverage this, –O2 aims to eliminate linear constraints by substituting a signal in these constraints with an equivalent linear expression, a process executed iteratively until no further simplification is possible for linear constraints containing at least one private signal. Specifically, the lazy Gaussian elimination algorithm operates in multiple rounds, identifying linear constraints and attempting to eliminate signals through substitutions. The unique aspect of this "laziness" lies in the fact that after a certain number of rounds and the application of some substitutions, initially non-linear constraints may transform into linear ones. The algorithm continues with further Gaussian elimination rounds until no new linear constraints are identified or a stopping condition is reached. Figure 2 illustrates a simple case of constraint-level optimization, where an intermediate "Signal C" is private; in such scenarios, the compiler can inline the expression to reduce the number of constraints.

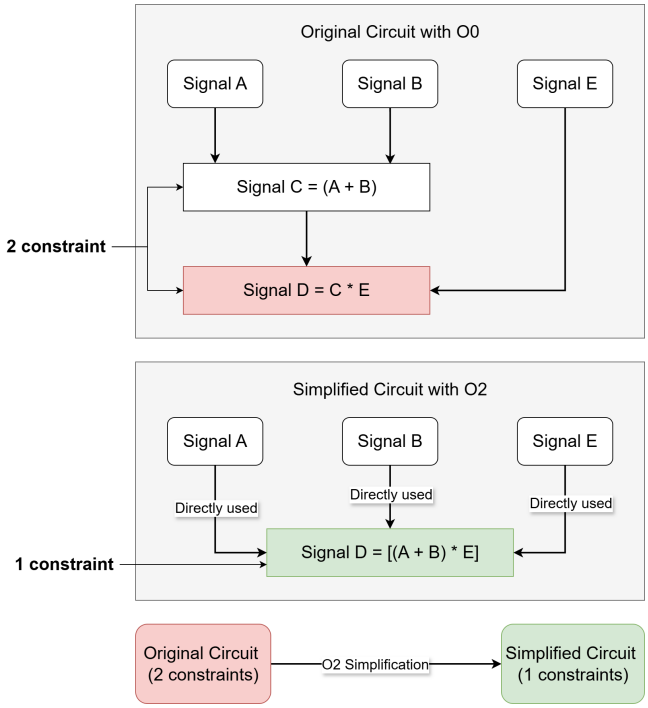

FIGURE 2. Illustration of the constraint simplification performed by Circom when using the –O2 optimization flag.

C. ANALYSIS OF –O2 OPTIMIZATION LEVEL

Through experiments, we observed that using the –O2 optimization flag of the Circom compiler leads to a significant increase in circuit compilation time compared to –O0 and –O1 flags. More specifically, the study found that the time taken for Gaussian elimination loops is not the most time-consuming stage in the –O2 optimization process; rather, the initial Gaussian elimination step before applying the loops consumes more time. To understand the reason, we analyzed the source code of the Circom compiler and found that the main difference comes from the large volume of linear constraints that the Gaussian elimination step in –O2 needs to process. Below is a detailed analysis:

1) Core Differences in Optimization Process

The core distinction between –O1 and –O2 is that the latter introduces an additional linear optimization phase. The –O1 level applies only basic simplification through direct substitutions, without complex elimination of linear constraints, as described in Section III-B1. In contrast, –O2 employs a cluster-based linear simplification algorithm, which significantly increases compilation time due to the large number of linear constraints that must be processed.

2) General Processing Stages and Costs

Both –O1 and –O2 perform a set of common preprocessing steps, whose computational cost grows significantly with circuit complexity—particularly because the outputs of these steps feed into the deeper optimization stages of –O2. The process begins with building the set of related signals, in which the compiler traverses the circuit's internal graph to identify all signals that participate in non-linear constraints and their dependencies. This traversal determines which signals must be preserved during simplification. Next, simple equality constraints are eliminated. These include assignments such as "signal = k" (where k is a constant) and "signal1 = signal2", which are replaced to reduce redundancy. Finally, the compiler rebuilds the set of related signals based on the updated constraint graph, recalculating connections after substitutions have been applied. This step ensures that the subsequent optimization phases operate on an accurate and minimal set of active signals.

3) Advanced Optimization Activities of –O2

The significant increase in compilation time observed at the –O2 optimization level is primarily attributed to several computationally intensive activities triggered by its linear optimization mechanism. One of the most critical processes is the cluster-based linear simplification. In this step, the compiler applies Gaussian elimination with parallel processing, grouping related constraints into clusters that share common signals. Although parallelism improves the efficiency of individual loops, the extremely large volume of constraints in ZK-Rollup circuits still leads to considerable processing time. –O2 executes multiple optimization rounds. Rather than applying transformations once, the compiler iteratively performs simplification passes until no further linear constraints can be eliminated. This requires repeated evaluation of all constraints and, importantly, tracking when previously non-linear constraints become linear as a result of earlier substitutions.

Another major contributor to the overall cost is the substitution and constraint reprocessing step. During each optimization loop, substitutions are applied across the entire constraint set, after which the system must re-examine all constraints to identify newly simplified ones. This continuous traversal and rechecking of constraints significantly increases

compilation time, especially for large circuits. The loop control mechanism in the compiler sets a limit on the number of optimization iterations. However, empirical observations show that the initial optimization loop consumes the most time, primarily due to the size of the linear constraint set that serves as the initial input.

The findings demonstrate a fundamental trade-off within Circom's compiler architecture: aggressive linear simplification reduces proving time at the expense of significantly increased compilation time. This trade-off is particularly acceptable in deployment scenarios where circuits are compiled once and reused many times, such as in prover-as-a-service infrastructures.

IV. METHODOLOGY

A. EXPERIMENTAL METHODOLOGY

1) Experimental Description

This study is designed as a quantitative experiment to evaluate the impact of Circom's constraint optimization flags on the performance of a ZK-Rollup system simulating ERC-20 transactions. Based on the findings, the study aims to propose a guideline framework for selecting appropriate constraint optimization flags for ZKP circuits (ZCLS).

The primary parameters in this study include the constraint optimization levels of Circom (–O0, –O1, –O2) and the transaction batch sizes (4, 8, 16, 32, and 64 transactions). The ZK-Rollup system implemented for the experiment incorporates core functionalities such as depositing assets into the layer 2 (ZK-Rollup), transferring funds within the rollup, and withdrawing funds back to layer 1. The experiment primarily focuses on assessing the performance of the batch transaction circuit, as it represents the most complex component encapsulating the core functionality of the ZK-Rollup system. Moreover, it provides a clearer indication of the computational and efficiency demands associated with processing multiple transactions concurrently.

To evaluate performance, two key zero-knowledge circuits were designed as part of the experimental setup. The first is the batch transaction circuit, which generates ZKPs for transaction batches processed off-chain. These proofs allow the layer 1 blockchain to verify the correctness of multiple transactions without executing them individually. The number of constraints in this circuit varies significantly depending on the number of transactions included in each batch. The second circuit is the withdrawal circuit, which enables users to withdraw assets from the layer 2 environment back to layer 1. This circuit exhibits a relatively stable number of constraints, with only minor variations depending on the maximum batch size that the system supports.

The implementation of the simulated ZK-Rollup system introduced here, along with the CirMetrics application described in subsequent sections, is available in the GitHub repository ZCLS-for-CirMetrics[1].

[1]https://github.com/datachain-uit/ZCLS-for-CirMetrics

2) Hardware Configuration

The experiments were conducted on a system with the following specifications:

- CPU: AMD Ryzen 8-core, 16-thread processor
- RAM: 32GB
- Operating system: Ubuntu 22.04 LTS

3) Metrics Used

To evaluate system performance, several core metrics were recorded and analyzed.

- The number of constraints was a primary indicator. This includes both the total number of constraints in the circuit and their breakdown into linear and non-linear categories. Since the number of constraints directly affects the computational cost and time for generating ZKPs, this metric is essential for understanding the complexity and efficiency of a given circuit.
- Compilation time refers to the duration required by the Circom compiler to translate the high-level circuit description into R1CS and associated files. This metric is especially relevant during the development phase of ZK circuits, as it impacts both turnaround time and development workflows.
- Proving time was measured using the Snarkjs library and indicates the time needed to generate a zero-knowledge proof for a transaction batch. Lower proving times translate into higher throughput for the ZK-Rollup system, as transactions can be finalized more quickly.
- Verification time denotes the time taken to validate the proof after it has been generated, both in off-chain and on-chain environments. This metric reflects how quickly transactions can be confirmed on layer 1, which is critical for end-user experience and application responsiveness.
- Gas consumption on the layer 1 blockchain was also recorded. This measures the computational resources (in gas units) required to verify a proof on-chain, and has direct implications for the economic efficiency of the ZK-Rollup protocol.
- Peak RAM usage was monitored during both the compilation and proving phases. This was measured using the Max Resident Set Size (Max RSS), indicating the maximum amount of memory consumed. This metric is particularly significant for resource-constrained environments such as prover-as-a-service models or embedded ZKP deployments.
- Data for these metrics was collected for each tested optimization flag and transaction batch size.

4) Number of Experimental Runs and Error Control

To ensure the reliability and accuracy of the experimental results, each measurement for compilation time, proving time and RAM usage for the two tasks mentioned was repeated multiple times. The experiment was conducted on a single hardware configuration with a focus on comparative analysis

of optimization levels, each configuration was executed at least 10 times. The average value of these repetitions was used as the official result for that configuration. This approach aligns with established practices in ZKP performance evaluation studies [17], where controlled experimental conditions allow for reliable data collection with a modest number of repetitions. Repeating the measurements helped minimize the influence of random factors and system noise, such as varying system load or background processes, which could potentially distort time-based results.The formula for calculating the average value is expressed as follows: $\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$

5) Analysis Plan

The collected data will be analyzed quantitatively to achieve the research objectives. The analysis consists of three main components. First, we will examine the relationship between constraint count and performance metrics, particularly focusing on how the number of constraints correlates with both compilation time and proof generation time. This analysis is intended to clarify how constraint simplification impacts computational cost at different stages of the zero-knowledge workflow. Second, we will evaluate trade-off factors between compilation time—which reflects the development cost of the application—and proof generation time, which impacts operational efficiency in production. Based on the experimental results, we will propose a multi-circuit optimization framework, tailored to the different stages of development. This framework will provide practical guidance for ZK-Rollup developers in selecting Circom optimization flags based on specific project priorities (e.g., the speed of iteration during development versus throughput requirements in a production environment) and the characteristics of each circuit.

B. SYSTEM ARCHITECTURE

1) Libraries and Tools Used

To implement and test the ZK-Rollup system, this study utilizes the following key libraries and tools.

- Hardhat (version 2.25.0) serves as a development environment for blockchain applications, particularly smart contracts on Ethereum, Hardhat provides tools for compiling, testing, and deploying smart contracts. It allows for the creation of local networks and integrates with libraries such as Ethers.js, facilitating a smooth development process. In this experiment, Hardhat supports the management and utilization of smart contracts and serves as a local layer 1 blockchain environment, simulating the Ethereum network to efficiently deploy and interact with smart contracts during development and testing.
- Circom (version 2.2.2) is a Circuit Description Language and compiler that enables the definition and writing of ZK circuits in an intuitive manner. Circom converts these circuits into the R1CS format, preparing them for the proof generation process.
- SnarkJS (version 0.7.5) is a JavaScript library used for proof generation and proof verification for ZK circuits. SnarkJS works in conjunction with Circom to complete the processes of proof creation and validation.
- Solidity (version 0.8.2) is the programming language for smart contracts, used to write the smart contracts for the rollup on layer 1. These contracts manage the state of the rollup and receive and verify ZKP proofs from off-chain sources.

2) ZK-Rollup Components

The ZK-Rollup system is designed to perform off-chain transactions and utilizes ZKPs to verify the accuracy of transactions when updating the state on the main chain. The system architecture consists of three main components: the sequencer, the circuit layer for proof generation, and the on-chain verifier.

The detailed process is illustrated in Fig. 4 and can be summarized with Fig. 3 as follows: The full ZK-Rollup transaction flow begins when users interact with the system. Token deposits are performed by invoking the rollup smart contract on Layer 1, which emits an event that the sequencer monitors to update the off-chain state accordingly. In contrast, fund transfers between accounts and withdrawal requests are submitted directly to the sequencer. The sequencer collects and queues these transactions until a predefined batch threshold is met. Once a transaction batch or group of withdrawal requests is ready, the sequencer prepares the necessary inputs for the zero-knowledge circuit, including Merkle proofs, digital signatures, and account states before and after each transaction. Using SnarkJS, the sequencer generates a ZKP along with the corresponding public signals. These outputs are then submitted to the appropriate smart contract on the layer 1 blockchain for verification. Upon successful validation of the proof, the rollup smart contract updates the Merkle tree to reflect the new on-chain state. Simultaneously, the sequencer finalizes and records the off-chain state updates, ensuring consistency across both layers of the system.

Sequencer (Off-chain State Management) The sequencer serves as the core component responsible for managing the off-chain state of the ZK-Rollup system. It performs several critical functions that enable secure and efficient batching of transactions before on-chain verification. First, it initializes and maintains two Merkle trees: the account tree, which stores user account states including public keys and balances, and the transaction tree, which records information about batched transactions. When a user deposits funds into the rollup system, the sequencer processes the deposit by updating the account tree and storing the updated root. As new transactions are submitted by users, the sequencer collects them, verifies their validity by checking account balances and digital signatures, and updates the account states accordingly. These validated transactions are then inserted into the transaction tree. Once a predefined batch threshold is met, the sequencer generates Merkle proofs and other required data that will later serve as inputs to the zero-knowledge circuit. This includes Merkle proofs, transaction signatures, and the pre/post transaction states for each account involved.

Circuit Layer (Zero-Knowledge Proof Generation) The circuit layer is responsible for verifying the correctness of transaction batches and withdrawal requests through ZKPs. It is implemented using the Circom language and includes two specialized circuits tailored for each verification task. The batch transaction processing circuit verifies fund transfers between accounts within the rollup system. It accepts several inputs including the Merkle root, Merkle proof, digital signature, and account states before and after the transaction. The logic embedded in the circuit performs multiple checks: it verifies the authenticity of each signature using EdDSA with Poseidon hashing, ensures that sufficient balances exist for all transfers, confirms the existence of both sender and recipient accounts in the Merkle tree, and validates that the resulting state transitions are accurate. These checks are executed through the use of reusable templates within the Circom circuit structure. The withdrawal processing circuit is designed to validate user withdrawal requests from the rollup system back to the main chain, which is emulated using the Hardhat local blockchain. It operates by verifying the digital signature authorizing the withdrawal, checking the account balance for sufficiency, confirming the existence of the account and transaction within the Merkle tree, and ensuring that the withdrawal destination corresponds to the zero address, which is a security measure for withdrawal confirmation.

On-chain Verification (Smart Contract Layer) On the layer 1 blockchain, the system relies on two types of smart contracts to perform verification and enforce state updates based on ZKPs. The primary contract is the rollup smart contract, which is responsible for maintaining the Merkle roots of both the account and transaction states. This contract receives proofs and public signals from the off-chain sequencer and invokes the appropriate verification contract to authenticate the submitted proof. Upon successful verification, it proceeds to update the on-chain state by processing deposits, internal transfers, and withdrawals. Additionally, the contract interacts with ERC-20 token standards to facilitate token movement between layer 1 and layer 2. Complementing this is the proof verification contract, which is automatically generated by SnarkJS based on the structure of the zero-knowledge circuit. This contract performs proof verification using the Groth16 protocol. State updates on the blockchain are permitted only if the submitted proof is successfully verified, ensuring that all critical operations, such as batched transfers and withdrawals, are cryptographically secured and resistant to fraud.

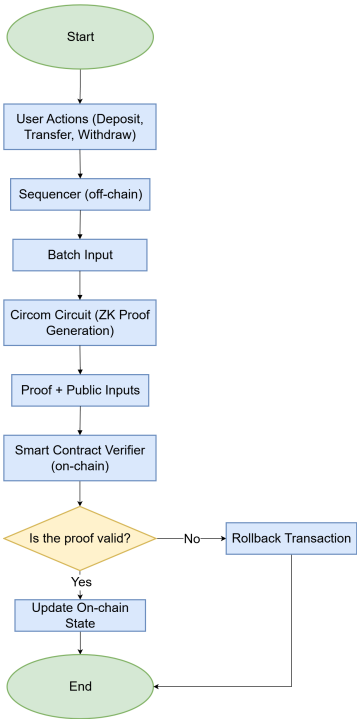

FIGURE 3. Diagram providing a simple description of the functions operating within ZK-Rollup simulation.

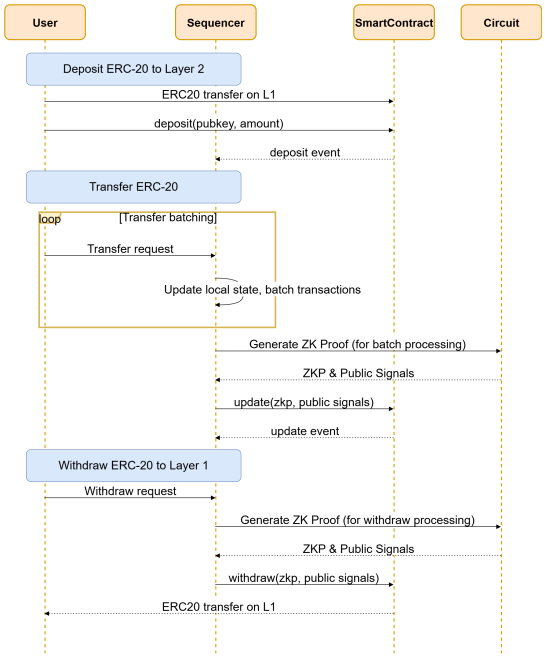

FIGURE 4. Diagram providing a detailed description of the operational functions within the ZK-Rollup simulation.

C. PROPOSED METHOD: ZK CIRCUIT LIFECYCLE STRATEGY (ZCLS)

Experimental results emphasize that compilation time is a significant development bottleneck for large-scale ZK-Rollup deployments. While it is necessary to use –O2 to ensure efficient proof generation, the trade-off with increased compilation time can disrupt the application development workflow. Therefore, the appropriate selection of circuit compiler constraint optimization flags plays a crucial role and needs careful consideration. Based on empirical analyses of the impact of Circom's –O0, –O1, and –O2 flags, we propose a structured methodology, called ZCLS. The ZCLS method provides a guiding framework for developers to make in-

formed decisions about ZKP circuit optimization throughout different stages of the development and deployment process.

ZCLS is an approach to manage and optimize the ZKP circuit compilation process, from initial design to product deployment. The goal of ZCLS is to balance development speed (compilation time) and execution performance (proof generation time), based on the specific requirements of each stage. ZCLS operates with the understanding that no single optimization flag is "best" for all situations; instead, the optimal choice depends on factors such as circuit update frequency, proof generation volume, and the current project stage.

1) Deciding Factors in ZCLS

ZCLS provides optimization recommendations by analyzing three primary factors: the development stage, circuit update frequency, and proof generation volume. Each of these factors influences the trade-offs between compilation speed and proof generation efficiency, thereby guiding the choice of optimization flags during different phases of the circuit development lifecycle. These thresholds are intended as guidelines rather than strict rules, offering developers a practical reference to align optimization strategies with circuit lifecycle dynamics.

The development stage is the most foundational and influential factor in the decision-making process. In the development and testing phase, circuits are frequently modified as new features are added and bugs are resolved. In this stage, rapid feedback is essential to maintain developer productivity. Consequently, the highest priority is minimizing compilation time, which enables faster iteration and shortens development cycles. Proof generation performance remains a relevant metric for functional testing, but it is not the dominant concern. When transitioning to the production deployment phase, the circuit becomes stable, and major changes are infrequent. Here, the focus shifts toward maximizing proof generation performance, especially in applications that require high throughput or real-time responsiveness. Since the frequency of circuit updates is greatly reduced, longer compilation times are acceptable and considered a reasonable trade-off for achieving lower proof latency and better runtime scalability. There also exists a special case for debugging, where developers aim to isolate or test small components of the circuit. In such situations, compilation speed becomes the absolute priority, even at the expense of proof generation efficiency. The objective is to rapidly check the correctness or behavior of specific components without waiting for a full compilation of the entire circuit or optimal proof generation. This use case justifies the use of low-optimization flags like –O0.

The second factor considered by ZCLS is the circuit update frequency, which refers to how often developers modify the circuit's source code during the development process. This factor is critical because it directly affects the time developers spend waiting for recompilation. A high update frequency demands fast compilation to avoid interrupting the development workflow, whereas a low update frequency allows for more expensive compilation stages if they lead to better runtime performance. In the case of high-frequency updates, developers modify circuits repeatedly (e.g., multiple times an hour or several times throughout the day, especially during active iteration and testing phases). Under these conditions, the cost of slow compilation becomes unacceptable, and strategies that minimize compilation time should be prioritized. Suggested quantitative thresholds for this case range from 10 to 50 compilations per day, reflecting active phases where rapid iteration is essential. In contrast, low-frequency updates typically apply to mature, production-ready circuits that are modified infrequently (e.g., only during protocol upgrades or security patches). In such cases, compilation time is less critical since changes are rare. Suggested thresholds for this scenario are less than once per week, such as monthly or quarterly updates. A very high-frequency debugging context may also arise when developers experiment with circuit logic or tune parameters intensively (e.g., recompiling multiple times per hour). Here, fast turnaround is crucial, and proof performance is secondary. Suggested thresholds exceed 50 recompilations per day, with compilation speed prioritized above all.

The third and final factor is the proof generation volume, which reflects the number of ZKPs expected to be generated during testing or deployment. This factor helps determine whether optimization should lean toward faster proof generation time, especially for large batches of transactions, or allow for slower proofs when volume is low. In high-volume scenarios, this volume arises in cases where proofs need to be generated for large transaction batches or when continuous proof generation is required to test system throughput. It is in these situations that optimizing proof generation time, such as with the –O2 flag, becomes extremely important. The requirement for proof generation time in this context will be rapid, with suggested quantitative thresholds exceeding 100 proofs per day. In low-volume environments, which refer to cases where proofs are generated infrequently or for small transaction batches during testing, the requirement for proof generation time will be moderate. Suggested quantitative thresholds range from 50 to 100 proofs per day. Finally, in very low-volume proof generation scenarios for debugging, where proof generation time is not a concern, absolute priority is given to compilation speed, even if proof generation performance is not optimal. This corresponds to the performance of –O0, which requires a very fast compilation time. Suggested quantitative thresholds are less than 1 proof per hour (e.g., 1 proof per day or fewer).

2) ZCLS Recommendation Table and Framework

Based on qualitative factors and corresponding performance requirements, the ZCLS recommendation table is presented in Table 2 and Fig. 5. The recommendations in the Fig. 5 and Table 2 are built upon the trade-off between compilation time and proof generation time, quantitatively derived from our benchmark data (especially at batch size 64, representing

TABLE 2. Recommended Circom compiler optimization levels based on development stage, circuit update frequency, and proof generation volume.

Development Stage	Circuit Update Frequency	Proof Generation Volume	Recommended Flag
Special Case (Debug)	Very High (Frequent)	Very Low	–O0
Development, Testing	High	Low	–O1
Development, Testing	High	High	–O1
Pre-Production, Production	Low	High	–O2
Pre-Production, Production	Low	Low	–O2

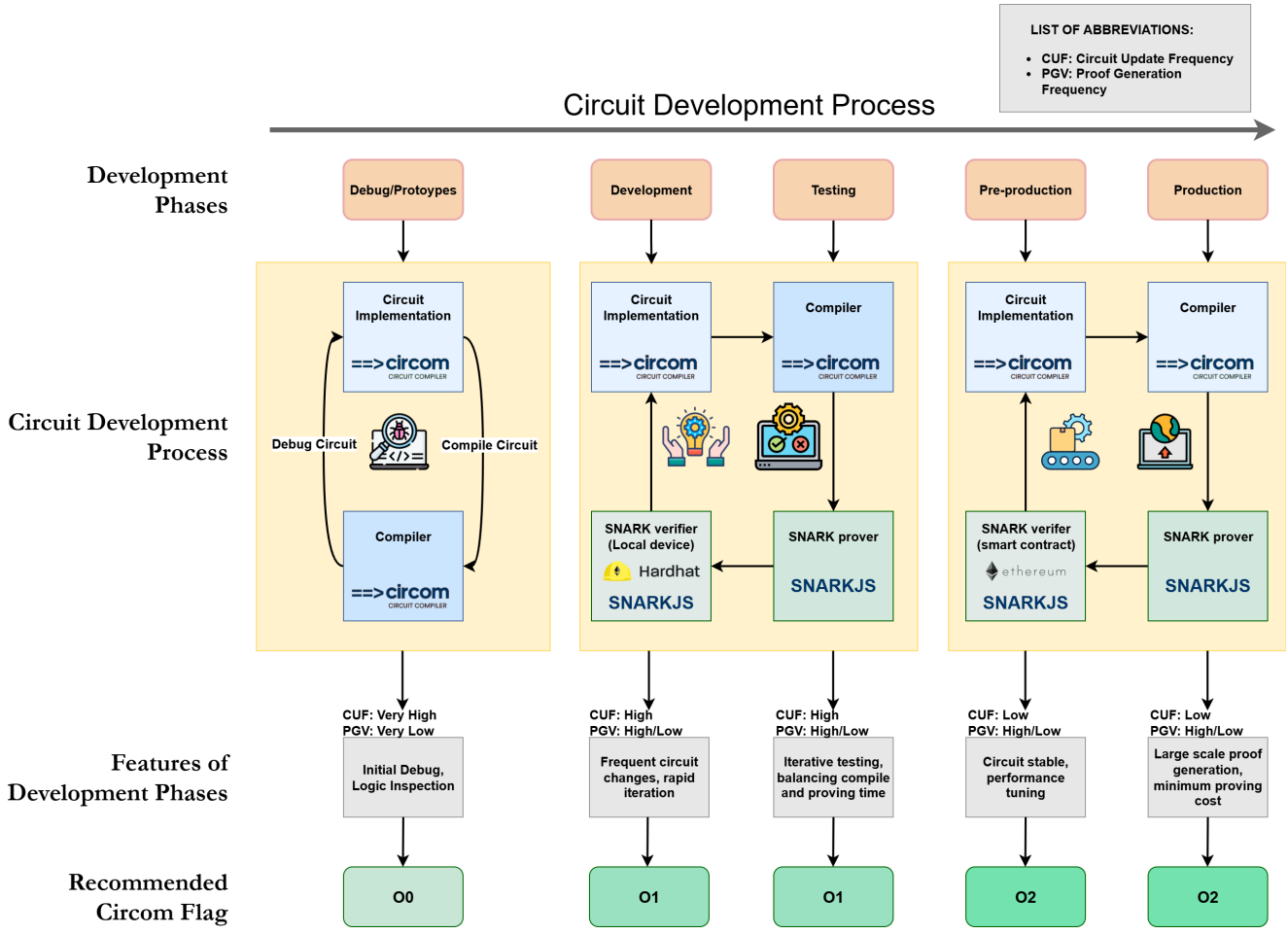

FIGURE 5. Flag selection framework for optimizing the Circom compiler based on development stages, frequency of circuit updates, and proof generation workload

the highest batch scenario in the current experiment). Below is an explanation for each specific case:

In the development and testing stage, circuits are frequently updated, and the compilation-feedback loop is critical for developer productivity. When the circuit update frequency is high and the proof generation volume is low, the top priority is fast compilation. In this scenario, –O1 is recommended as it offers a reasonable balance, with compilation time at approximately (~66s) and proving time at (~35s). While –O0 compiles faster, its proving time is significantly slower at (~78s), which may hinder testing efficiency. On the other hand, –O2 achieves better proving performance at (~28s), but its compilation time is excessively long at (~125s), making it unsuitable for rapid development iterations. Even when the proof generation volume is high, such as during stress testing or high-load simulations, –O1 continues to provide a practical and balanced solution. It maintains a fast compilation time (~66s) while delivering sufficiently fast proof generation performance (~35s), enabling developers to test circuit behavior under load without experiencing significant delays due to

compilation overhead.

In the context of production deployment, the circuit is typically stable and rarely updated. Therefore, longer compilation times can be tolerated in favor of runtime performance improvements. When the proof generation volume is high, as in real-world ZK-Rollup deployments handling large batches of transactions, the top priority shifts to maximizing throughput. –O2 is the optimal choice under these conditions, delivering the fastest proving time (~28s), outperforming both –O1 (~35s) and –O0 (~78s). The increased compilation time of –O2 (~125s) becomes acceptable, given that compilation occurs infrequently in production environments. Even if the proof generation volume is low, –O2 remains the preferred option. In production, maximizing efficiency is still desirable to ensure system stability and responsiveness. While other flags may offer marginally faster compilation, the consistent proving performance of –O2 makes it a robust choice for production use. This reflects a fundamental principle of system deployment: every opportunity for performance optimization contributes meaningfully to long-term reliability.

Finally, in special debugging scenarios, debugging sessions often require very high circuit update frequency, as developers frequently recompile to test small changes. In such cases, minimizing compilation time becomes the overriding concern. –O0 is the most suitable flag, offering the fastest compilation time (~48s), which significantly accelerates the debugging loop. Although its proving time is the slowest among the three options (~78s), this is not a limiting factor, as full proof generation is rarely necessary during debugging. The main goal is to quickly identify and resolve logic issues, and –O0 is best equipped to support this requirement.

3) ZCLS Application Process

The ZCLS method will help developers make informed optimization decisions, thereby contributing to enhancing the efficiency of development and performance of applications using ZKP like ZK-Rollup. To apply ZCLS in practice, developers can follow these steps: (1) Identify the current Development Stage: Is the project in the Development and Testing, Production Deployment, or Debugging stage? (2) Assess Circuit Update Frequency: Determine if you are updating the circuit with "High," "Low," or "Very High" frequency. (3) Assess Proof Generation Volume: Determine if you need to generate proofs with "High," "Low," or "Very Low" volume. (4) Refer to the ZCLS Recommendation Table: Based on the assessments in steps 1, 2, and 3, consult the recommendation table to find the appropriate optimization flag. (5) Make adjustments (if necessary): If the developer is working with a circuit or environment whose scale differs significantly from the empirical data of this study, the developer can prioritize "Circuit Update Frequency" and "Proof Generation Volume" and choose the flag that best suits the specific real-world environment. The Circom optimization flag selection guide remains valuable in this case. (6) Repeat the process: ZCLS is a continuous cycle. As the project moves to different stages or requirements change, developers should repeat the evaluation process to adjust the optimization flags accordingly.

4) Practical Implementation: CirMetrics Tool

While ZCLS provides a strategic framework for selecting constraint optimization flags, practical adoption in real-world development scenarios benefits from tooling support. To facilitate this, we introduce CirMetrics—a dedicated analysis and visualization tool that supports developers in applying ZCLS effectively during ZK circuit design. CirMetrics is designed to extract and monitor circuit performance metrics in real time, enabling developers to compile Circom circuits with different optimization flags and retrieve detailed constraint statistics, which include the total number of constraints along with their breakdown into linear and non-linear categories, compilation time under various optimization levels, and proof generation time tracked across circuit versions and input data sets. CirMetrics offers interactive visualization capabilities, providing comparative charts that allow developers to observe changes in constraint distributions as optimization strategies are applied, track the evolution of proof generation time across circuit iterations, compare trade-offs between optimization levels for different circuit scales, and visualize the frequency of circuit modifications and proof generations during development cycles.

By integrating empirical metrics with ZCLS decision-making logic, CirMetrics bridges the gap between theoretical guidelines and day-to-day development practice. Instead of automating optimization decisions, the tool empowers developers to make informed choices by combining real-time diagnostics with lifecycle-aware recommendations. This integration enhances ZK circuit maintainability, reduces trial-and-error cycles, and improves overall performance planning. A preview of CirMetrics' analysis, results view and dashboard interfaces is shown in Figures 6, 7, and 8.

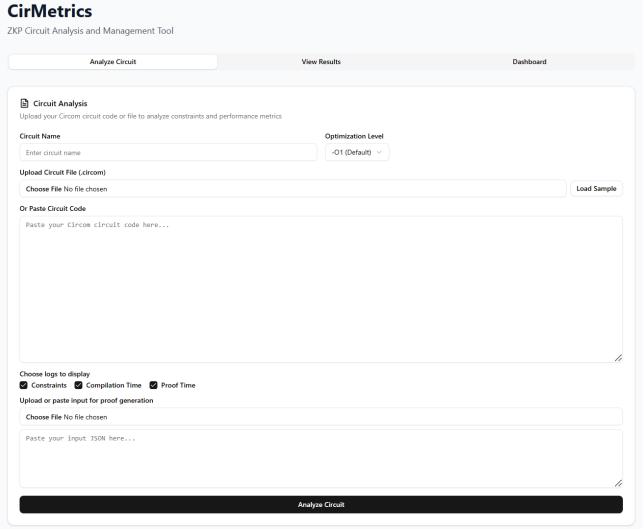

FIGURE 6. Circuit Analysis Feature

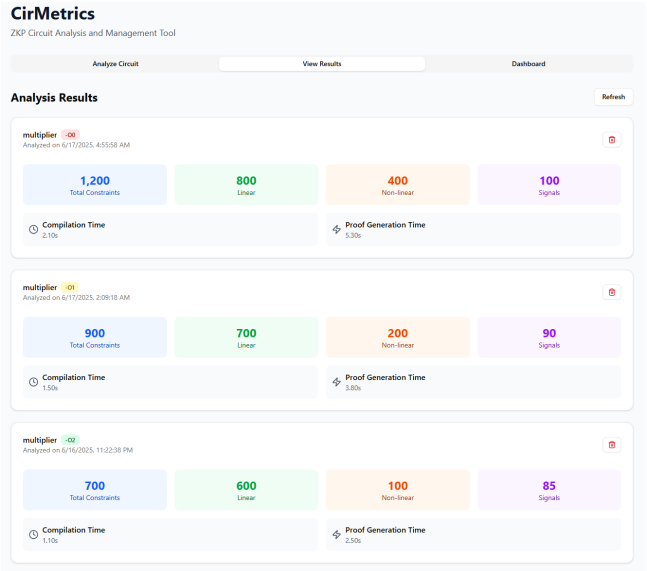

FIGURE 7. Analysis Results Feature

FIGURE 8. Summary Chart for ZK Circuit Parameters

V. EXPERIMENTAL RESULTS AND EVALUATION

A. EXPERIMENTAL RESULTS

Table 3 presents a comprehensive summary of experimental measurements across five batch sizes (4, 8, 16, 32, and 64) under three Circom compiler optimization levels: –O0, –O1, and –O2. The results include the number of non-linear and linear constraints, total constraint count, and the ratio of linear to total constraints. Additionally, compilation time, proving time, and their relative changes compared to the –O0 baseline are reported. Memory usage during both compilation and proving stages is also recorded to assess resource requirements. This consolidated view enables systematic analysis of the relationship between constraint optimization and system performance across different circuit scales and optimization strategies.

B. DISCUSSION

1) Analysis of the impact of constraint optimization levels on recorded metrics (RQ1)

The results in Table 3 reveal distinct performance profiles for each Circom optimization level across varying batch sizes. At the –O0 level, circuits exhibit a high volume of linear constraints, consistently constituting around 67% of the total constraints regardless of batch size. This indicates a substantial presence of structurally redundant or algebraically trivial relations, which the compiler does not attempt to eliminate at this level. In contrast, –O1 introduces moderate constraint simplification, reducing the linear-to-total constraint ratio to approximately 41–42%. The –O2 level applies the most aggressive optimization, consistently removing all linear constraints across all configurations and yielding the lowest total constraint counts.

This reduction in total constraints under –O2 correlates strongly with improved proving performance. For instance, at batch size 64, the total number of constraints decreases from 3,623,746 (–O0) to 969,728 (–O2), accompanied by a drop in proving time from 77.6 seconds to 27.7 seconds, representing a 64.3% reduction. Similar trends are observed at lower batch sizes, where –O2 continues to outperform both –O0 and –O1 in terms of proof generation latency. The results demonstrate that eliminating linear constraints, which do not contribute meaningfully to the computational logic of the circuit, can significantly accelerate zk-SNARK proving without sacrificing correctness or completeness.

Nevertheless, these gains come with notable trade-offs. Compilation time under –O2 increases substantially, more than doubling in most batch configurations compared to –O0. At batch size 64, compilation time rises from 47.7 seconds under –O0 to 125.4 seconds under –O2. Similarly, peak memory consumption during compilation increases by a factor of nearly 8, from 0.58 GB to 4.49 GB. These costs are attributable to the computational complexity of the –O2 pipeline, which performs iterative simplification and cluster-based elimination strategies to exhaustively reduce the constraint system. This behavior reflects a fundamental shift in performance distribution, where the burden is intentionally moved from proof generation to circuit preprocessing.

Importantly, while linear constraints are entirely removed by –O2, non-linear constraints remain largely stable across optimization levels. At batch size 64, the number of non-linear constraints only drops by approximately 18% when comparing –O2 to –O0. This is consistent with the nature of ZK-Rollup circuits, where most cryptographic operations, such as signature verification, Merkle proof verification, and hash functions, contribute inherently non-linear constraints that are essential to preserving circuit semantics.

TABLE 3. Analysis of Compilation Time, Proving Time, and Peak RAM Usage across Different Batch Sizes and Compiler Optimization Levels

Batch Size	Flag	Constraints				Time (ms)		Change compare to O0		Memory Usage (GB)	
		Non-linear Constraints	Linear Constraints	Total Constraints	Ratio (L/T)	Compilation	Proving	Compile Time Overhead (%)	Proving Time Reduction (%)	Maximum Compilation	Maximum Proving
4	O0	69972	143874	213846	0.67	5712	6259			0.39	4.50
	O1	69912	48432	118344	0.41	6973	4032	22.07	35.58	0.43	3.18
	O2	56720	0	56720	0	12934	3166	126.44	49.42	0.60	2.56
8	O0	141936	292074	434010	0.67	8289	11988			0.44	6.53
	O1	141816	99104	240920	0.41	11197	7059	35.08	41.12	0.43	4.30
	O2	115384	0	115384	0	18018	5106	117.37	57.41	0.74	3.13
16	O0	287856	592802	880658	0.67	12873	27685			0.46	12.60
	O1	287616	202688	490304	0.41	18923	14296	47.00	48.36	0.53	6.44
	O2	234656	0	234656	0	39184	8629	204.39	68.83	1.13	4.40
32	O0	583680	1202914	1786594	0.67	25861	36874			0.49	15.25
	O1	583200	414336	997536	0.42	33010	30705	27.64	16.73	0.98	11.54
	O2	477088	0	477088	0	62516	20211	141.74	45.19	2.24	7.22
64	O0	1183296	2440450	3623746	0.67	47731	77601			0.58	28.49
	O1	1182336	846592	2028928	0.42	65744	35413	37.74	54.36	1.88	17.52
	O2	969728	0	969728	0	125411	27686	162.75	64.32	4.49	10.80

Furthermore, the observed improvements in proving time under –O2 do not translate into any change in proof size or verification gas cost. This stability stems from the properties of the Groth16 proving system, which generates a fixed-size proof comprising three elliptic curve elements [30]. Verification costs are similarly unaffected, as they scale with the number of public inputs rather than constraint complexity. Therefore, constraint optimization primarily benefits the prover and has limited impact on on-chain verification, making it particularly advantageous in high-throughput environments.

In conclusion, constraint optimization levels in Circom produce measurable and significant effects on proving performance and compilation overhead. The –O2 level delivers the highest reduction in proof generation time by aggressively eliminating linear constraints, though at the expense of increased preprocessing time and memory usage. –O1 provides a middle ground with moderate gains and lower resource demands. These trade-offs underscore the importance of selecting optimization levels based on system scale, proof volume, and deployment context, validating the motivation for a formalized strategy.

2) Experimental scenarios to assess the impact of ZCLS on ZK-Rollup development performance (RQ2)

We utilized the experimental data collected to simulate the effects of ZCLS in realistic hypothetical scenarios:

Scenario 1: Intensive Debugging Phase. A developer is debugging a complex ZK circuit with a batch size of 64 and needs to make frequent updates during a single workday. The circuit is modified around 100 times per day, and recompilation is required after each change. Proof generation is minimal and only used to validate correctness after basic circuit updates. The main objective here is to minimize the total compilation time, thereby enabling a rapid feedback loop essential for productive debugging.

This use case corresponds to the ZCLS condition of very high circuit update frequency combined with very low proof generation volume. According to our benchmark data at batch size 64, the average compilation time was measured at 47.731 seconds for –O0, 65.744 seconds for –O1, and 125.411 seconds for –O2. Using these figures, the total compilation time for 100 recompilations per day would be approximately 79.6 minutes for –O0, 109.6 minutes for –O1, and 209 minutes for –O2. The difference is significant: adopting –O0 results in a 27.4% time reduction compared to –O1 and a 61.7% reduction compared to –O2. These results support the ZCLS recommendation of using –O0 in debugging phases, where compilation time is critical and proof performance is less important. By reducing wait time during recompilation, developers maintain momentum and reduce friction in the debugging workflow.

Scenario 2: Production Deployment with High Transaction Volume. A ZK-Rollup system is already deployed in production and handles thousands of transactions daily. The circuit is considered stable and rarely updated. The dominant concern is minimizing proof generation time to achieve high throughput and operational efficiency. The goal in this case is to maximize proving performance, as compilation is infrequent and can tolerate higher latency.

This setting aligns with the ZCLS profile of low circuit update frequency and high proof generation volume. Our benchmark data show that proof generation times at batch

size 64 average 77.601 seconds for –O0, 35.413 seconds for –O1, and 27.686 seconds for –O2. Assuming 1000 proofs are generated per day, the total proving time would be roughly 21.6 hours for –O0, 9.8 hours for –O1, and just 7.7 hours for –O2. Compared to –O1, –O2 saves over two hours daily. Compared to –O0, the reduction exceeds 14 hours. This confirms the suitability of –O2 in production deployments where system throughput is paramount. Although –O2 incurs higher compilation costs, these costs are offset by the benefits realized over long production cycles. Choosing –O2 significantly boosts scalability, reduces operational delays, and aligns well with the long-term efficiency goals of production ZK-Rollup systems.

3) Scalability and adaptability of ZCLS to real-world systems (RQ3)

We acknowledge that the current experiments are conducted with a maximum batch size of 64. In practice, ZK-Rollup systems can handle transaction batches of up to several thousand or even tens of thousands, resulting in a significant increase in the number of constraints and compilation/proof generation times. Therefore, the values used in the framework analysis should be understood as based on the data within the scope of our experiments. However, the core principles and trends regarding the trade-offs between compilation time and proof generation time across optimization flags (–O0, –O1, –O2) will still hold true at larger scales. Specifically:

- –O0 will be the fastest option for compilation but the slowest for proof generation.
- –O2 will be the slowest option for compilation but the fastest for proof generation.
- –O1 will provide a reasonable balance between these two factors.

This framework offers a structured approach to help developers make informed decisions regarding optimization flags in Circom, ensuring that the chosen strategy aligns with the specific requirements of the application development stage. By providing empirical data and a proposed framework for selecting optimization flags, this study extends theoretical discussions on ZKP performance, offering practical guidance for developing decentralized applications (DApps) and ZK applications in real-world scenarios. The proposed optimization framework enables developers to manage the complexity of ZKP generation, minimize trial and error time, and accelerate application deployment.

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

This study conducted a comprehensive empirical analysis of the impact of Circom compiler optimization flags (–O0, –O1, –O2) on the performance of ZK-Rollup systems processing ERC-20 transactions, using the Groth16 backend. Through simulation and evaluation of key metrics such as the number of constraints, circuit compilation time, proof generation time and peak RAM usage for those two tasks mentioned, the study clarified the performance trade-offs associated with each optimization level.

The results show that the –O2 optimization flag, despite significantly increasing circuit compilation time, delivers superior efficiency in reducing the number of constraints and substantially shortening proof generation time. This confirms the value of deep optimization at the constraint level for the performance of the proving process, which is a critical bottleneck in ZKP systems. Conversely, the –O1 flag provides a reasonable balance, reducing approximately 56% of constraints compared to –O0 without incurring excessive compilation costs, indicating it is a suitable choice for iterative development and testing phases. The study also reconfirmed that, for Groth16, proof size and proof verification time are stable and independent of the number of constraints, while proof generation time is directly affected by the number and nature of constraints.

Based on these quantitative analyses, we proposed the practical ZCLS optimization flag selection framework, which helps developers make informed decisions based on the project development stage, circuit update frequency, and proof generation volume. This framework not only optimizes resource utilization but also contributes to accelerating the development and deployment of applications using ZKP in practice.

This study provided valuable detailed empirical data to validate theoretical predictions related to constraint system optimization, along with a structured methodology to better understand the operational mechanisms of optimization options in Circom. This can not only help developers save time and resources when developing and deploying ZK-Rollup applications but also support the widespread adoption of zero-knowledge proof technology in blockchain systems and other applications requiring high security and performance.

Despite efforts to achieve the stated research objectives and provide detailed quantitative analyses of the impact of optimization flags in the Circom compiler, our study still has certain limitations. Experiments were conducted on an AMD Ryzen 8-core, 16-thread system with 32GB RAM, ensuring consistency but potentially yielding different results on other hardware architectures or more complex environments. The simulated ERC-20 circuit chosen as a case study, while complex enough to evaluate constraint optimization, does not fully reflect the diversity of real-world ZK-Rollup systems with various transaction types and complex smart contracts. Experimental results on small batches (≤ 64) cannot accurately predict behavior with thousands of transactions. However, these results provide an overview of trends and optimization mechanisms, serving as a starting point for understanding behavior at a larger scale. Additionally, the scope of the study was limited to the Circom compiler and the Groth16 zk-SNARK system, without comparison to other ZKP systems such as Plonk. Nevertheless, focusing on a specific set of tools allowed for more detailed analysis, yielding important insights into constraint optimization, and will serve as a foundation for many future research directions.

B. FUTURE WORK

To further develop and expand this research, we propose several potential directions. These development directions will not only deepen the understanding of ZKP optimization but also contribute to building more robust, efficient, and user-friendly ZK-Rollup systems in the future.

- Comparison with other ZKP compilers: Extend the research to compare constraint optimization capabilities and overall performance between Circom and other ZKP compilers such as ZoKrates, Noir, or Rust/Halo2-based frameworks. This will provide a more comprehensive view of the ZKP optimization landscape and help developers choose the most suitable tool for their needs.
- Investigate advanced optimization strategies both at the circuit design level, such as restructuring arithmetic logic or redesigning constraint flows to eliminate redundancy and improve constraint efficiency, and within compiler internals, including enhancements to constraint simplification algorithms beyond existing flag-based heuristics.
- Evaluation on other transaction types and applications: Expand the experimental scope to include more complex transaction types (e.g., DeFi transactions, NFT minting) or other ZK-Rollup applications beyond ERC-20 token transfers. This will help assess the generality and applicability of the optimization results.
- Development of automation and integration tools: Develop tools or plugins to automate the flag selection process based on the input parameters of the circuit and desired performance objectives. Additionally, integrate research findings into existing ZK-Rollup development tools to facilitate the seamless adoption of the proposed recommendations.

ACKNOWLEDGMENT

This research was supported by The VNUHCM-University of Information Technology's Scientific Research Support Fund.

REFERENCES

[1] H. Song, Z. Qu, and Y. Wei, "Advancing blockchain scalability: An introduction to layer 1 and layer 2 solutions," in *2024 IEEE 2nd International Conference on Sensors, Electronics and Computer Engineering (ICSECE)*, pp. 71–76, IEEE, 2024.

[2] Etherscan, "Pending transactions chart." https://etherscan.io/chart/pendingtx. Accessed: Jun. 5, 2025.

[3] Etherscan, "Token transactions list." https://etherscan.io/tokentxns. Accessed: Jun. 5, 2025.

[4] S. Chaliasos, I. Reif, A. Torralba-Agell, J. Ernstberger, A. Kattis, and B. Livshits, "Analyzing and benchmarking zk-rollups," in *6th Conference on Advances in Financial Technologies (AFT 2024)*, pp. 6–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[5] L. T. Thibault, T. Sarry, and A. S. Hafid, "Blockchain scaling using rollups: A comprehensive survey," *IEEE Access*, vol. 10, pp. 93039–93054, 2022.

[6] M. B. Saif, S. Migliorini, and F. Spoto, "A survey on data availability in layer 2 blockchain rollups: Open challenges and future improvements," *Future Internet*, vol. 16, no. 9, p. 315, 2024.

[7] L. Zhou, A. Diro, A. Saini, S. Kaisar, and P. C. Hiep, "Leveraging zero knowledge proofs for blockchain-based identity sharing: A survey of advancements, challenges and opportunities," *Journal of Information Security and Applications*, vol. 80, p. 103678, 2024.

[8] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, "{SoK}: What don't we know? understanding security vulnerabilities in {SNARKs}," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 3855–3872, 2024.

[9] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4733–4751, 2022.

[10] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[11] 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.

[12] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[13] Y. Gong, Y. Jin, Y. Li, Z. Liu, and Z. Zhu, "Analysis and comparison of the main zero-knowledge proof scheme," in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, pp. 366–372, IEEE, 2022.

[14] J. Liang, D. Hu, P. Wu, Y. Yang, Q. Shen, and Z. Wu, "Sok: Understanding zk-snarks: The gap between research and practice," *arXiv preprint arXiv:2502.02387*, 2025.

[15] J. L. Munoz-Tapia, M. Belles, M. Isabel, A. Rubio, and J. Baylina, "Circom: A robust and scalable language for building complex zero-knowledge circuits," *Authorea Preprints*, 2023.

[16] M. El-Hajj and B. Oude Roelink, "Evaluating the efficiency of zk-snark, zk-stark, and bulletproof in real-world scenarios: A benchmark study," *Information (Switzerland)*, vol. 15, no. 8, p. 463, 2024.

[17] 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," in *International Conference on Security and Cryptography for Networks*, pp. 46–72, Springer, 2024.

[18] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, "Automated detection of under-constrained circuits in zero-knowledge proofs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1510–1532, 2023.

[19] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, and A. Rubio, "Distilling constraints in zero-knowledge protocols," in *International Conference on Computer Aided Verification*, pp. 430–443, Springer, 2022.

[20] J. Liu, L. Guo, and T. Kang, "Genes: An efficient recursive zk-snark and its novel application in blockchain," *Electronics*, vol. 14, no. 3, p. 492, 2025.

[21] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Annual cryptology conference*, pp. 90–108, Springer, 2013.

[22] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 499–530, Springer, 2023.

[23] H. Lipmaa, "Polymath: Groth16 is not the limit," in *Annual International Cryptology Conference*, pp. 170–206, Springer, 2024.

[24] X. Tang, L. Shi, X. Wang, K. Charbonnet, S. Tang, and S. Sun, "Zero-knowledge proof vulnerability analysis and security auditing," *Cryptology ePrint Archive*, 2024.

[25] H. Chen, G. Li, M. Chen, R. Liu, and S. Gao, "Ac4: Algebraic computation checker for circuit constraints in zkps," *arXiv preprint arXiv:2403.15676*, 2024.

[26] F. Dahlgren, "It pays to be circomspect." https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 2022. Accessed: 2025-06-02.

[27] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng, "Certifying zero-knowledge circuits with refinement types," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 1741–1759, IEEE, 2024.

[28] E. Foundation, "Erc-20 token standard." https://ethereum.org/vi/developers/docs/standards/tokens/erc-20/, 2024. Accessed: 2025-06-02.

[29] A. Ballesteros-Rodríguez, S. Sánchez-Alonso, and M.-Á. Sicilia-Urbán, "Enhancing privacy and integrity in computing services provisioning using blockchain and zk-snarks," *IEEE Access*, 2024.

[30] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques,*

Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35, pp. 305–326, Springer, 2016.
[31] X. Zhu, X. Zhang, X. Song, Y. Deng, Y. Wei, and L. Yang, "Extending groth16 for disjunctive statements," *Cryptology ePrint Archive*, 2025.
[32] J. Partala, T. H. Nguyen, and S. Pirttikangas, "Non-interactive zero-knowledge for blockchain: A survey," *IEEE Access*, vol. 8, pp. 227945–227961, 2020.
[33] C. Shi, H. Chen, R. Liu, and G. Li, "Data-flow-based normalization generation algorithm of r1cs for zero-knowledge proof," in *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 191–197, IEEE, 2023.
[34] J. Eberhardt and S. Tai, "Zokrates-scalable privacy-preserving off-chain computations," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1084–1091, IEEE, 2018.
[35] J. C. Díaz Rodríguez, "Extending the circom compiler," *Docta Complutense, Complutense University of Madrid*, 2023.

KHOA TAN VO , a Master of Science in Information Technology from the University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM), is a lecturer of the Faculty of Information Science and Engineering at VNU-HCM. Specializing in Software Engineering and Blockchain Technology, he actively contributes to various research projects and has a robust portfolio of publications in prestigious international conferences and journals. His dedication to advancing technology and education underscores his commitment to excellence in his field. You can reach him at khoavt@uit.edu.vn or n240203@grad.uit.edu.vn.

MINH NGO , is an undergraduate student in Information Technology at the University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM). His academic interests lie in the intersection of blockchain systems, zero-knowledge proofs, and software engineering. He is particularly interested in building decentralized applications and exploring the design of scalable, privacy-enhancing technologies. You can reach him at 21521129@gm.uit.edu.vn.

THU NGUYEN , with a Master of Science in Computer Science from the University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM), is a lecturer of the Faculty of Information Science and Engineering at VNU-HCM. Her research interests lie in Data Science, AI, and Machine Learning. She has participated in numerous projects and published papers in prestigious international conferences and journals, demonstrating her commitment to advancing these fields. You can reach her at thunta@uit.edu.vn or n240204@grad.uit.edu.vn.

THU-THUY TA holds a Master of Science in Computer Science from the University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM). She is a lecturer of the Faculty of Information Science and Engineering at VNU-HCM. Her research focuses on Data Science and Machine Learning, areas in which she has contributed to several projects and published papers in international conferences and journals. Her work reflects a commitment to advancing the fields of data science and machine learning. You can reach her at thuyta@uit.edu.vn.

MONG-THY NGUYEN THI holds a Master of Arts in Teaching English as a Second Language (TESL) from Benedictine University, USA. She is an educator at the Center for Foreign Languages, University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM). Her research focuses on English Language Teaching (ELT) Methodology. Passionate about improving language education, she contributes significantly to advancing ELT practices at her institution. You can reach her at thyntm@uit.edu.vn.

HONG-TRI NGUYEN is a postdoctoral researcher at Aalto University. Before Aalto, he received a B.S. degree in computer science from the University of Information Technology – Vietnam National University, Vietnam, in 2015, and an M.S. degree in computer science from the University of Pisa, Italy, in 2018. Also, he obtained a Ph.D at the University of Oulu in 2023. His research interests include distributed systems, blockchain technology, and information security.

TU-ANH NGUYEN-HOANG , Ph.D., is an Associate Professor at the University of Information Technology, Vietnam National University, Ho Chi Minh City (VNU-HCM). She earned her Ph.D. in the Mathematical Foundations of Computer Science and Computational Systems from the University of Science, VNU-HCM. Her research focuses on Data Mining, Soft Computing, and Machine Learning. Dr. Tu-Anh is actively involved in numerous research projects and has a prolific publication record in international conferences and journals. Her work significantly contributes to advancing these specialized fields. You can reach her at anhnht@uit.edu.vn.