

Titre: Securing Ethereum's Ecosystem: Formal Verification and Abstract Interpretation of Halo2 Circuits and Innovative Access Control of Smart Contracts
Title:

Auteur: Fatemeh Heidari Soureshjani
Author:

Date: 2024

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Heidari Soureshjani, F. (2024). Securing Ethereum's Ecosystem: Formal Verification and Abstract Interpretation of Halo2 Circuits and Innovative Access Control of Smart Contracts [Thèse de doctorat, Polytechnique Montréal].
Citation: PolyPublie. <https://publications.polymtl.ca/59285/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/59285/>
PolyPublie URL:

Directeurs de recherche: Hanifa Boucheneb
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**Securing Ethereum's Ecosystem: Formal Verification and Abstract
interpretation of Halo2 Circuits and Innovative Access Control of Smart
Contracts**

FATEMEH HEIDARI SOURESHJANI

Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Août 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

**Securing Ethereum's Ecosystem: Formal Verification and Abstract
interpretation of Halo2 Circuits and Innovative Access Control of Smart
Contracts**

présentée par **Fatemeh HEIDARI SOURESHJANI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Frédéric CUPPENS, président

Hanifa BOUCHENEB, membre et directrice de recherche

Nora CUPPENS, membre

Hakima OULD SLIMANE, membre externe

DEDICATION

To my family, especially my parents, who always have unyielding love and support. And to my brother Mehrdad, whose compassionate heart we have lost, but his memory remains with us...

ACKNOWLEDGEMENTS

This dissertation is supervised by Dr. Hanifa Boucheneb. I would like to express my most sincere appreciation to Dr. Boucheneb for her patience, wise comments, advice, precious assistance, and valuable support and for providing the opportunity to discover a new field of state-of-the-art technologies.

Besides all the support from my advisor, I cannot ignore my family's unconditional love and generous support. I would like to dedicate my dissertation to my amazing father and my mom for her continued love.

RÉSUMÉ

Dans le paysage en constante évolution de la technologie blockchain, l'émergence des applications décentralisées (DApps) a révolutionné notre vision de la confiance, des opérations et des services dans un monde numérique.

L'augmentation du déploiement des DApps et du volume des transactions a mis en lumière le problème de l'scalabilité. Ethereum, la plateforme principale pour les DApps, a rencontré des défis pour évoluer tout en maintenant la décentralisation et la sécurité. Des solutions telles que les zk-Rollups ont été proposées pour atténuer ces contraintes en regroupant ou en roulant plusieurs transactions en une seule preuve. Il existe également de nombreuses applications de la preuve à divulgation nulle de connaissance (Zero Knowledge) dans les protocoles offrant des transactions préservant la vie privée sur la blockchain comme Zcash. La sécurité de ces technologies repose sur la fiabilité des systèmes de preuve à connaissance zéro sous-jacents. Étant donné la complexité et la nature critique de ces systèmes, les méthodes de vérification formelle sont primordiales pour garantir que les systèmes de preuve sont correctement implémentés et que les circuits internes sont robustes face aux vulnérabilités.

En outre, cœur de ce changement de paradigme se trouve le besoin de mécanismes de sécurité robustes régissant le contrôle d'accès et maintenant la sécurité des contrats intelligents.

Dans ce contexte, cette recherche vise à contribuer de manière substantielle au domaine de la sécurité de la blockchain.

Cette dissertation se lance dans deux domaines de recherche révolutionnaires : la vérification formelle et interprétation abstraite des systèmes de preuve à connaissance zéro, avec un focus sur les circuits Halo2 et le Contrôle d'Accès Basé sur Blockchain en tant que Service.

Le premier segment est pionnier dans la vérification formelle des circuits Halo2, essentiels pour les protocoles préservant la vie privée et les zk-Rollups, utilisant des solveurs SMT pour détecter les circuits sous-contraints et introduisant une nouvelle méthode formelle légère pour analyser l'arithmétisation basée sur PLONK. Significativement, cette recherche a produit le premier outil jamais créé pour analyser les circuits Halo2, en créant initialement un analyseur dynamique qui modélise le système de contraintes de Halo2 pour une vérification avec SMT solver, et par la suite en affinant la conception vers une version entièrement statique qui optimise l'efficacité du solveur SMT. Cette exploration non seulement avance le cadre théorique pour la sécurité de la blockchain et les preuves à connaissance zéro mais fournit également des outils pratiques et des méthodologies pour améliorer la fiabilité et la performance de ces tech-

nologies de pointe. De plus, nous proposons une amélioration de l'interprétation abstraite des circuits halo2 pour détecter les éléments inutilisés dans un circuit qui peuvent causer plusieurs problèmes de sécurité comme une vulnérabilité sous-contrainte. Le second segment introduit une plateforme innovante simplifiant le contrôle d'accès dans les contrats intelligents à travers un modèle de Contrôle d'Accès Centré sur les Rôles, combinant la simplicité du Contrôle d'Accès Basé sur les Rôles (CABR) avec la flexibilité du Contrôle d'Accès Basé sur les Attributs (CABA). Ce modèle améliore le contrôle d'accès au niveau des fonctions et permet des mises à jour dynamiques des politiques basées sur les attributs, permettant ainsi aux développeurs de se concentrer sur les fonctionnalités essentielles sans sacrifier la sécurité.

Par ces deux voies, cette recherche contribue à la discussion critique et au développement de la sécurité de la blockchain, en abordant le rôle central des systèmes sans connaissance dans l'évolutivité d'Ethereum et la question urgente du contrôle d'accès aux contrats intelligents.

ABSTRACT

In the ever-evolving landscape of blockchain technology, the emergence of decentralized applications (DApps) has revolutionized how we envision trust, operations, and services in a digital world.

The surge in DApp deployment and transaction volume has brought the issue of scalability and privacy to the fore. Ethereum, the leading platform for DApps, has faced challenges in scaling while maintaining decentralization and security. Solutions such as zk-Rollups have been proposed to alleviate these constraints by bundling or rolling up transactions into a single proof. There are also numerous applications of Zero knowledge in protocols providing privacy-preserving transactions on blockchain like Zcash. The security of these technologies hinges on the reliability of the underlying zero-knowledge-proof systems. Given these systems' complexity and critical nature, formal verification methods are paramount to ensure that the proof systems are correctly implemented and that the circuits within are robust against vulnerabilities.

In addition, at the core of this paradigm shift is the need for robust security mechanisms that govern access control and maintain the security of smart contracts.

In this context, this research aims to contribute substantially to the blockchain security field.

This dissertation ventures into two groundbreaking research domains: the formal verification and abstract interpretation of zero-knowledge proof systems, with a focus on Halo2 circuits, which are one of the most popular implementations of Plonk, and Blockchain-Based Access Control as a Service. The first segment pioneers the formal verification of Halo2 circuits, critical for privacy-preserving protocols and zk-Rollups, using SMT solvers to detect under-constrained circuits and introducing a novel, lightweight formal method for analyzing PLONK-based arithmetization. Significantly, this research produced the first-ever tool for analyzing Halo2 circuits, initially creating a dynamic analyzer that models Halo2's constraint system for verification with SMT solver, and subsequently refining the design to a fully static version that optimizes SMT solver efficiency. In addition, we propose an improvement for the abstract interpretation of halo2 circuits for detecting unused elements within a circuit which can cause several security issues like under-constrained vulnerability. The second segment introduces an innovative platform that simplifies access control in smart contracts through a Role-Centric Access Control model, blending the simplicity of RBAC with the flexibility of ABAC. This model enhances function-level access control and enables dynamic updates of attribute-based policies, thereby allowing developers to concentrate on core functionalities

without sacrificing security. This dual-focus exploration not only advances the theoretical framework for blockchain security and zero-knowledge proofs but also delivers practical tools and methodologies for enhancing the reliability and performance of these cutting-edge technologies.

Through these two avenues, this research contributes to the critical discussion and development of blockchain security, addressing the pivotal role of zero-knowledge-proof systems in Ethereum's scalability and the pressing issue of smart contract access control.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
LIST OF TABLES	xiii
LIST OF FIGURES	xv
LIST OF SYMBOLS AND ACRONYMS	xvii
CHAPTER 1 INTRODUCTION	1
1.1 Automated Analysis of Halo2 Circuits: Motivation and contribution	1
1.2 Studying Role-Centric Access Control as a Service on Blockchain: Motivation	3
1.3 Objectives and Contributions	5
1.3.1 Objectives	5
1.3.2 First Contribution: Automated analysis of Halo2 Circuits	6
1.3.3 second Contribution: Static Halo2 Analyzer and Cycle Notion-Based Optimization	6
1.3.4 Third Contribution: Abstract Interpretation for PLONKish Circuits	6
1.3.5 Fourth Contribution: Role-Centric Access Control as a Service	6
1.4 Organization of the Thesis	7
CHAPTER 2 BACKGROUND	9
2.1 Principles of cryptography	9
2.1.1 Symmetric and Asymmetric Cryptography	9
2.1.2 Cryptographic Hash Functions	11
2.2 Blockchain	13

2.2.1	Peer-to-Peer Networks	13
2.2.2	Distributed Ledger Technology	14
2.2.3	Blockchain Structure	14
2.2.4	Consensus Mechanism	16
2.2.5	Ethereum	18
2.3	Zero-Knowledge Proofs: PLONK and SNARKs in Ethereum’s Ecosystem . .	19
2.3.1	Fundamentals of SNARKs	21
2.3.2	Arithmetization in Zero-Knowledge Proofs	23
2.3.3	Utilization of Tables in Halo2	25
2.4	Ethereum Scalability	26
2.4.1	Ethereum 2.0	26
2.4.2	Layer 2	26
2.4.3	Zero-Knowledge Ethereum Virtual Machines (zkEVMs): Enhancing Privacy and Scalability in Ethereum	27
2.5	Satisfiability Modulo Theories (SMT) Solvers	28
2.6	Access Control	29
2.6.1	Role-Based Access Control: RBAC	30
2.6.2	Attribute-Based Access Control: ABAC	30
2.6.3	RBAC vs ABAC	31
2.6.4	Combining RBAC and ABAC: Role-Centric Access Control	31
2.7	Conclusion	32
CHAPTER 3	RELATED WORK AND LITERATURE REVIEW	34
3.1	Tools for Analyzing Proof Systems	34
3.1.1	Domain-Specific Languages for Zero-Knowledge Proofs	35
3.1.2	Comprehensive Comparison of Zero-Knowledge Proof Analysis Tools	37
3.2	Blockchain-Based Access Control Management for On-Chain DApps	37
3.3	Blockchain-Based Access Control Management for Off-Chain Platforms	39
CHAPTER 4	AUTOMATED ANALYSIS OF HALO2 CIRCUITS	41
4.1	Introduction	41
4.2	Halo2	43
4.3	Under- and Over-constrained Circuits	45
4.4	From Halo2 to SMT	47
4.5	Analyzer Design and Implementation	51
4.6	Fuzzing	54
4.7	General Property Checking	57

4.8	Discussion	57
CHAPTER 5 HALO2 ANALYZER: STATIC DESIGN AND CYCLE NOTION-BASED OPTIMIZATION		
		59
5.1	Introduction	59
5.2	Halo2 Circuits and the Analyzer	60
5.3	Analyzer Design	60
5.4	Optimization Technique Based on Permutation Argument	62
5.4.1	Understanding Permutation and Cycles	62
5.4.2	Permutation Schema in Halo2	62
5.4.3	Optimization Implications for the SMT Solver	64
5.4.4	Practical Example: Fibonacci Circuit	65
5.5	Evaluation	67
5.5.1	Benchmark Setup	67
5.5.2	Benchmark Results	68
5.6	Conclusion	68
CHAPTER 6 ABSTRACT INTERPRETATION FOR PLONKISH CIRCUITS		
		70
6.1	Introduction	70
6.2	Related Work	72
6.3	Preliminaries	73
6.3.1	PLONKish Arithmetization	73
6.3.2	Abstract Interpretation	75
6.3.3	Prior use of Abstract Interpretation	75
6.4	Improvements	76
6.4.1	Adding Support for Copy Constraints	76
6.4.2	Adding Support for Lookups	76
6.5	Conclusion	79
CHAPTER 7 BLOCKCHAIN-BASED ROLE-CENTRIC ACCESS CONTROL AS A SERVICE		
		81
7.1	Blockchain-Based Role-Centric Access Control as a Service	81
7.1.1	Role-Centric Access Control Model for Smart Contracts	82
7.1.2	Privacy-Preserving Access Control using Blockchain	85
7.1.3	Design Alternatives	89
7.1.4	Proposed Design Solution	91

7.1.5	Implementation: Ethereum-Based Role-Centric Access Control as a Service	92
7.2	Evaluation	94
7.2.1	Methodology	94
7.2.2	Metrics	94
7.2.3	Discussion	95
7.3	Scalability of Role-Centric Access Control as a Service Platform	96
7.3.1	Benchmark Setup	96
7.3.2	Results	98
7.3.3	Overall Scalability	99
7.3.4	Considerations for Large-Scale Implementations	99
7.4	Use Case: Managing Margin in Perpetual Protocol	100
7.4.1	Introduction: The Perpetual Protocol	100
7.4.2	The Current Issue	100
7.5	Conclusion and Future work	103
CHAPTER 8	CONCLUSION	104
REFERENCES		107

List of Tables

Table 3.1	Overview of Tools for Zero-Knowledge Proof Analysis	37
Table 4.1	A very small Halo2 circuit's table, filled with real values. The corresponding polynomial constraint is equation (4.1). The table is filled to demonstrate that the prover knows an assignment $a = 5$ such that $C \cdot a = 3 \cdot 5 = 15 = b$ (which implies $C \cdot a - b = 0$, as required). That is, the prover knows the necessary factor a such that $C \cdot a = b$. Since $s = 1$, the gate is on.	44
Table 4.2	A small range check modelled as a lookup in a Halo2 circuit table (only the relevant columns are shown). In this case, we want the i row of the advice column A_0 , a_i to be one of the values in the only fixed column F_0 . We add the following constraint to the solver: $(a_i = c89efdaa) \vee (a_i = 7f8b6b08) \vee \dots \vee (a_i = ceebf77a)$	50
Table 4.3	A lookup in a Halo2 circuit table with constraints on the fixed columns (only the relevant columns are shown). In this case, the entry of F_2 at row i is the bit-wise exclusive or function (XOR) applied to the values of row i in F_0 and F_1 . In this case, we want the i row of the advice columns A_0 , A_1 , and A_2 to be a valid XOR for entries which have three bits. We add the following constraint to the solver: $(a_{0,i} = 000 \wedge a_{1,i} = 000 \wedge a_{2,i} = 000) \vee (a_{0,i} = 000 \wedge a_{1,i} = 001 \wedge a_{2,i} = 001) \vee \dots \vee (a_{0,i} = 111 \wedge a_{1,i} = 111 \wedge a_{2,i} = 000)$	50
Table 4.4	Run times for the analysis in Section 4.3 on generalized circuits of example 4.3.1.	57
Table 5.1	Benchmark results comparing execution times of Version 1 and Version 2 of the halo2 analyzer on the Fibonacci circuits of various row sizes.	68

Table 6.1	An example of a Halo2 circuit's table, simplified. The table has a number of rows that span selector (S_i), advice (A_i), lookup (L_i^j) or fixed (F_i) (in fact, there may be multiple different lookup functions L^j , which we group for convenience), and instance (I_i) columns. Selector column values are either 0 or 1 while the remaining columns can range over the entire finite field. The values of the Selector columns are set at circuit configuration and the lookup or fixed columns are public constants known to both the prover and the verifier. The advice columns are private and known only to the prover and entered when the statement is to be proved. Solid arrows indicate a copy constraint between the relevant cells of the table, namely, both ends of the arrow should have the same value when the entire table is evaluated. Dashed arrows represent lookups. The red cells may be empty or unassigned when the table is populated; not every cell is necessarily used.	74
Table 7.1	Benchmarking results for two architectures. All costs are in gas units.	95
Table 7.2	Benchmarking results across two scenarios and two architectures. All costs are in gas units.	95

List of Figures

Figure 2.1	Symmetric Cryptography	10
Figure 2.2	Asymmetric Cryptography	10
Figure 2.3	Digital Signature	11
Figure 2.4	Cryptographic Hash Functions	11
Figure 2.5	(a)Calculate the hash of a data and store it (b) recalculate the hash of data and compare it with the stored hash	13
Figure 2.6	Client Server vs Peer-to-Peer network	14
Figure 2.7	Blockchain structure	15
Figure 2.8	Merkle Tree structure	15
Figure 2.9	Transaction Verification in Merkle Tree	16
Figure 2.10	Ali Baba Cave	20
Figure 2.11	Overview of Popular Zero-Knowledge Proofs (ZKPs), showcasing key interactive, non-interactive, zk-SNARK, and zk-STARK frame- works	21
Figure 2.12	Role-Based Access Control Model	30
Figure 2.13	Attribute-Based Access Control Model	31
Figure 4.1	The process used to detect under- and over-constrained systems in our tool.	49
Figure 4.2	Analyzer Design	54
Figure 5.1	The redesigned halo2 Analyzer showcasing the integration of the Analyzable structure and cycle notion-based optimization.	61
Figure 5.2	Tracking a sample cycle Via Assembly of a Circuit: (a) Copy constraints (b) Assembly.sizes Matrix (c) Assembly.mapping Matrix .	64
Figure 5.3	Permutation argument for the Fibonacci circuit.	66
Figure 6.1	The evaluation table for bitwise AND on 3-bit numbers (a) and how to abstractly evaluate it (b).	80
Figure 7.1	Access Control as a Service.	81
Figure 7.2	User's attributes encryption and validation	88
Figure 7.3	Factory Design Pattern	90

Figure 7.4	Proposed Design	92
Figure 7.5	Gas Cost vs Number of ABACs	98
Figure 7.6	Gas Cost vs Number of ABACs	99

LIST OF SYMBOLS AND ACRONYMS

DApp	Decentralized Application
IoT	Internet of Things
RBAC	Role-Based Access Control
ABAC	Attribute-Based Access Control
OrBAC	Organization Based Access Control
P2P	Peer to Peer
PoW	Proof of Work
PoS	Proof of Stake
EOA	Externally Owned Accounts
EVM	Ethereum Virtual Machine
zkEVM	Zero-Knowledge Ethereum Virtual Machine
ZKP	Zero-Knowledge Proof
SNARK	Succinct Non-Interactive Argument of Knowledge
ZK-Rollup	Zero-Knowledge Rollup
CP-ABE	Ciphertext-Policy Attribute-Based Encryption
DeFi	Decentralized Finance
UTXO	Unspent Transaction Output
DSL	Domain Specific Language
eDSLs	Embedded Domain Specific Language
R1CS	Rank-1 Constraint System

CHAPTER 1 INTRODUCTION

This thesis explores two important areas within the security of the broader scope of blockchain technology and cryptography: security of zero-knowledge proofs and access control in blockchain-based decentralized applications. The inherent complexities and security sensitivities associated with these areas make them significant research avenues, as improved methods and practices can substantially impact the robustness and efficiency of blockchain systems. The focus of this research is the Ethereum blockchain as the pioneer platform of decentralized applications.

1.1 Automated Analysis of Halo2 Circuits: Motivation and contribution

The development of *zero-knowledge proof* systems, especially *zk-SNARKs* (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) [1–4], has significantly advanced the field of cryptography, offering profound implications for blockchain technologies and privacy-preserving digital applications. The popularity of zk-SNARKs lies in their three defining properties: being *zero-knowledge* to ensure verification without additional information disclosure, *succinctness* for efficient verification, and *non-interactivity* to streamline the proof process. Systems like Groth16 [5], Halo [6], and PLONK [2] exemplify zk-SNARKs’ capabilities. Prominent implementations include Tornado Cash [7], which provides privacy for Ethereum transactions, and zkRollups like zksync [8], Zircuit [9], scroll [10], and Polygon zkEVM [11], which significantly enhance Ethereum’s scalability by bundling multiple transactions into a single proof.

Significance of zk-SNARKs in zkEVMs The critical role of zero-knowledge proof systems, particularly zk-SNARKs, extends beyond general privacy and efficiency enhancements to foundational security implications for blockchain infrastructures like zkEVMs (zero-knowledge Ethereum Virtual Machines). zkEVMs leverage the power of zk-SNARKs to execute and verify Ethereum transactions off-chain in a scalable and secure manner. In this architecture, the entire operational security of the zkEVM depends fundamentally on the underlying zero-knowledge proof system—typically, a system like Halo2. Halo2’s robustness ensures that operations performed off-chain meet the same security standards as those executed on the Ethereum mainnet. This dependency places a significant emphasis on the reliability of Halo2 circuits. Any vulnerabilities or inefficiencies in these circuits could compromise the security and integrity of the zkEVMs, potentially leading to significant financial

and operational risks. Thus, the development and thorough analysis of such circuits are paramount, underlining the need for sophisticated tools and methods for automated analysis to identify and rectify potential weaknesses before deployment. This ensures that the systems not only uphold the privacy and scalability benefits but also maintain stringent security standards essential for widespread adoption and trust in blockchain technologies.

One of the critical parts of any ZKP system is arithmetization. Arithmetization is the process of transforming computational problems into mathematical forms that cryptographic proof systems can efficiently process. This transformation underpins SNARKs and other zero-knowledge proof systems, enabling them to handle intricate computations. Among the most prevalent forms of arithmetization is the Rank-1 Constraint System (R1CS) [12]. R1CS translates computations into a series of linear equations, each involving variables that are linearly combined and equated to zero.

Furthermore, Plonkish arithmetizations represent an evolution in this area, enhancing the traditional frameworks with optimizations aimed at minimizing proof sizes and the computational overhead of verifications. By employing polynomial representations and advanced mathematical techniques, Plonkish arithmetizations improve the efficiency of SNARK systems. Such advancements have dramatically increased the practical utility and performance of zero-knowledge proofs, particularly within blockchain technology where the demands for scalability are critical.

The potential of zk-SNARKs in blockchain scalability stems from their ability to succinctly verify computations off-chain, a cornerstone for solutions known as *rollups* [13]. These systems, leverage computational integrity proofs without necessitating knowledge privacy. zk-SNARKs facilitate the development of complex zero-knowledge applications through Domain-Specific Languages (DSLs) like Circom and ZoKrates, or Embedded Domain Specific Language (eDSLs) such as Halo2 [14], which simplify the arithmetization of computational functions for zero-knowledge proofs.

However, the complexities of zero-knowledge applications impose significant challenges, with potential bugs in DSL and eDSL implementations or the arithmetization process posing security risks. The development of automated tools is vital in such complex development environments. Several tools like Ecne [15], Picus [16], and QED [17] have been developed to ensure the security of Rank-1 Constraint Systems. These tools check if a circuit is *under-constrained*. Under-constrained vulnerabilities are one of the most critical security issues in ZK that could have allowed user funds to be stolen in Tornado Cash [18] or allowed repeated spending of tokens in the Aztec network [19].

Despite advancements in R1CS security tools, the lack of such tools for analyzing PLONKish

arithmetizations, especially Halo2 circuits, presents a critical research challenge and marks it as an open area for exploration. Addressing this, this thesis leverages the CVC5 SMT solver [20] and abstract interpretation [21] to introduce the first tool capable of detecting under and over-constrained Halo2 circuits. This development represents a significant step towards enhancing the robustness and security of zero-knowledge proof systems, thereby meeting the urgent need for reliable cryptographic tools in blockchain technologies.

1.2 Studying Role-Centric Access Control as a Service on Blockchain: Motivation

With the recent significant advancements of blockchain technology, we are witnessing the migration of various centralized applications to decentralized development platforms using blockchain [22]. Decentralized application development was first introduced on the Ethereum network, and to date, Ethereum has been the predominant platform for decentralized application development. These decentralized applications are developed using smart contracts [23]. The concept of the smart contract was first introduced by Szabo [24] as a set of digitally expressed promises and the protocols that oblige the parties to fulfill these promises. Ethereum has a virtual machine, a quasi-Turing complete state machine that executes these smart contracts' bytecodes. There are several high-level programming languages for developing these smart contracts, such as Solidity. These contracts are deployed on the blockchain, and users can execute the contracts' functions. These decentralized applications must guarantee the security of their data and resources. Blockchain technology allows all the executed and verified transactions or events to be kept immutable.

Immutability is a notable feature of smart contracts. This means updating them after deploying on the blockchain will be impossible. This can be considered a positive feature because users are confident that the contract rules will be the same forever. But this can be problematic in another way. If logical errors or vulnerabilities in the code are not detected before deployment, it is impossible to fix them. In addition, smart contracts often deal with transferring and maintaining cryptocurrencies, and a vulnerability can lead to huge financial losses. Smart contract vulnerabilities have led to various attacks in recent years and many coding patterns which cause these vulnerabilities have been identified that developers should avoid [25]. The DAO attack was the first major attack on Ethereum, in which the reentrancy vulnerability was exploited to drain more than 3.6 million ethers (\$60 million) from the DAO contract. The damage was so severe that to neutralize its impact, a consensus was formed in the network for a hard fork [26].

One of these vulnerabilities is related to the lack of proper access controls to the functions

of the smart contracts. The lack of a comprehensive access control system that meets the requirements of smart contract developers can lead to serious and complex problems. This leads to contracts with weak access control management, opening the door to several attacks. Parity Attack [27] was one of the attacks related to the lack of proper access control in which the contract owner had access to execute the self-destruct function (a function which deletes a contract from the blockchain). The attacker could use a library to become the contract owner, and by calling the self-destruct function, 513,774.16 ethers were blocked in the wallets. Another attack was on Poly Network [28], which took place in August 2021 and was described as the largest DeFi attack at the time. It was a very complex attack from a security point of view, in which the attacker or team of attackers put together several vulnerabilities in the contracts, which led to the theft of \$ 610 million from the network and its users. An address as a keeper had access to perform a function responsible for cross-chain transactions, and the hackers managed to change the keeper's address and execute this function. This emphasizes the importance of access control management of the functionalities of smart contracts. Our goal in this part is to explore and address the existing deficiencies.

The first notable problem is that there are only pure Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC) or Policy-Based Access Control models for smart contracts, each one alone having some limitations.

In addition, It is very common for developers to be assigned to define and manage access control policies within the core business logic of their decentralized applications (DApps). Hence, a common problem in smart contracts development is the tight coupling of access control management and business logic of smart contracts making the logic too complicated and prone to vulnerabilities. The most widely used libraries such as OpenZeppelin, while facilitating a basic role definition and management of a role-based access control model, leave the policy definition and management to the developers.

Moreover, this rigid development methodology leaves little room for updating access control system elements, such as roles, attributes, and policies. This poses significant challenges due to the immutable characteristics of deployed smart contracts. Some level of updatability has been proposed in several pieces of research, however, we have noticed that no existing solution provides a fully updatable platform supporting role-based and attribute-based models. There is no on-chain access control model that, at the time of deployment, requires no initial setup of the access control parameters like roles or attributes. This means that the owner of a protocol needs to have some knowledge of initial requirements and set them up at deployment time, for example, on the roles or the attributes which will be required in the future.

Another concern about the existing research in the field of smart contracts is the lack of

comprehensive discussions about design patterns. Specifically, platforms like access control management can be considered to have a unique logic like RBAC or ABAC; this is important to analyze all the possible design patterns and choose the design pattern which minimizes the gas cost, which is the most important performance metric in the Ethereum ecosystem.

One of the most interesting topics in Ethereum is having privacy-preserving protocols. On the other hand, when it comes to access control management, transparency of the policies will also have an important role. These two factors might contradict each other and are worth thorough discussion to understand the limitations and possibilities of designing privacy-preserving blockchain-based access control platforms.

In this research, we aim to propose solutions for these deficiencies via innovative designs and discuss the feasibility of solutions to some of these issues.

1.3 Objectives and Contributions

The overarching objective of this thesis is to enhance the security and scalability of the Ethereum ecosystem. This goal is pursued through two main directions. Firstly, automated tools for the Halo2 library. Halo2 libraries are widely used for building zkEVMs, which are the building blocks of Ethereum’s rollup-based scalability solutions. Second, the development of innovative frameworks for access control management in smart contracts. These efforts aim to secure the scaling of Ethereum through analysis of Halo2 circuits and to provide robust solutions for managing access control more effectively, thereby reducing vulnerabilities within Ethereum smart contracts. Herein, the specific objectives and their corresponding contributions are delineated:

1.3.1 Objectives

1. Enhance Automated Analysis of Halo2 Circuits: Develop and refine methodologies that improve the automated verification of circuit correctness using formal verification and abstract interpretation techniques.
2. Advance Access Control Management: Innovate and streamline access control mechanisms within Ethereum smart contracts to reduce complexity and enhance security and manageability.

1.3.2 First Contribution: Automated analysis of Halo2 Circuits

- We implement a proof of concept¹ for automatically checking such applications for correctness for the Halo2² proof system, which uses PLONKish arithmetization.
- We provide a method to check if a circuit is under-constrained. We employ a Satisfiability Modulo Theories (SMT) solver to try to find multiple satisfying models (i.e., private inputs) for a circuit and a (fixed) public input to determine if it is under-constrained.
- we also check if the circuit is over-constrained by Employing an SMT solver if there are no satisfying assignments to private and public inputs.

1.3.3 second Contribution: Static Halo2 Analyzer and Cycle Notion-Based Optimization

- We provide a different approach to improve the initial proof of concept tooling for detecting under-constrained circuits via a static analyzer design with a significant improvement in execution time via Cycle Notion-based Optimization.

1.3.4 Third Contribution: Abstract Interpretation for PLONKish Circuits

- We propose a new approach for the abstract interpretation of Halo2 circuits by considering the copy constraints and lookup constraints. This will increase the analysis accuracy by decreasing false negatives on detecting unused gates, columns, and cells, which can potentially make a circuit under-constrained.

1.3.5 Fourth Contribution: Role-Centric Access Control as a Service

- We introduce Role-Centric Access Control as a Service, a platform that effectively handles function-level access control in Ethereum smart contracts. This platform leverages the strengths of Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) to mitigate vulnerability threats while simplifying access control management.
- Our architecture requires minimal initial setup from developers and subsequently provides policy, role, user, and attributes management. This approach significantly reduces development burdens and mitigates risks of human errors, enhancing the security of decentralized applications (DApps).

¹Available at <https://github.com/quantstamp/halo2-analyzer>

²<https://github.com/zcash/halo2>

- We further discuss the impact of smart contract design patterns on gas efficiency and the trade-offs between user privacy and policy auditability in blockchain-based access control systems. We propose a method for on-chain storage of user attributes to protect privacy without compromising the integrity and auditability of access control policies.

1.4 Organization of the Thesis

The thesis is structured to provide a comprehensive understanding of blockchain technologies, focusing on access control in decentralized applications and automated analysis of Halo2 circuits. The organization of the thesis is as follows:

- **Chapter 1: Introduction** - This chapter introduces the key themes, objectives, and significance of the research. It sets the context and outlines the scope of the thesis.
- **Chapter 2: Background** - This chapter provides the necessary background information on Ethereum Blockchain ecosystem. It covers fundamental concepts and technologies that are critical for understanding the subsequent chapters.
- **Chapter 3: Related Work and Literature Review** - Here, a review of existing literature relevant to the thesis topic is presented. This chapter discusses previous research efforts and situates the current research within the broader academic context.
- **Chapter 4: Automated Analysis of Halo2 Circuits** - Focusing on Halo2 circuits, this chapter presents the methods and tools developed for automated error detection and analysis in Halo2 circuits.
- **Chapter 5: halo2 analyzer: static design and Cycle Notion-based Optimization** - Chapter 6 delves into the enhancement of the halo2 analyzer, building upon the foundations laid in Chapter 5. This segment critically evaluates previous design methodologies and introduces a novel, static design for the analyzer. This advancement paves the way for leveraging permutation-based optimizations, significantly increasing the tool's efficiency.
- **Chapter 6: Abstract Interpretation for PLONKish Circuits** - This chapter explores the application of abstract interpretation techniques to improve the accuracy of checks for unused gates, assigned but unconstrained values, and possibly unconstrained circuits in the PLONKish arithmetization used by the Halo2 proof system.

It highlights how copy constraints and arbitrary lookup functions are utilized in abstract interpretation-based analysis of these circuits, offering advancements in detecting potential bugs and vulnerabilities.

- **Chapter 7: Blockchain-Based Role-Centric Access Control as a Service** - This chapter delves into the development and implementation of a blockchain-based access control service. It discusses the challenges, solutions, and impacts of this service in the context of Ethereum smart contracts.
- **Chapter 8: Conclusion** - The final chapter concludes the thesis by summarizing the key findings, discussing the implications of the research, and suggesting potential directions for future work.

Each chapter builds upon the previous, methodically advancing the reader's understanding of blockchain technology, smart contract security, and automated analysis methods. This structured approach aims to provide a clear and coherent narrative throughout the thesis.

CHAPTER 2 BACKGROUND

In this chapter, we will have a brief review of the basic concepts on which our proposed platform will be built. These concepts are cryptography and its application, access control, blockchain, privacy-preserving access control using blockchain and its limitations, Zero-knowledge proof systems and their role in Ethereum scalability and privacy-preserving, Halo2 and PLONK, and finally, the usage of Halo2, PLONK, and SNARK in the Ethereum ecosystem.

2.1 Principles of cryptography

In this part, we will firstly explain symmetric cryptography and asymmetric cryptography. asymmetric cryptography is also known as public-key cryptography is one of the most applicable concepts used in blockchain to achieve security. It is also used to protect the privacy of users while storing their data on-chain. Secondly, we will explain Cryptographic Hash Functions which some of the most basic concepts of blockchain are built on them, like hash pointers, Merkle trees, etc.

2.1.1 Symmetric and Asymmetric Cryptography

Cryptography is the art of sharing a secret between two parties using some kind of secret key in a way that a third party who does not have access to this key is not able to access the secret. Assume a sender wants to send a Plain-Text PT to a receiver through an insecure channel, in which adversaries can see what is communicated. In the symmetric cryptography model, the sender uses a key known as secret key: SK to encrypt this document; i.e. the sender encrypts the document PT using SK . The encrypted document is known as Cipher-Text (CT): $CT = \text{Encrypt}(PT, SK)$. Then CT is sent to the receiver. The sender needs to send SK via a secure channel to the receiver, so the receiver will be able to decrypt the received encrypted document (CT): $PT = \text{Decrypt}(CT, SK)$. This process is shown in Figure 4.1. The problem is that, by the way, even if there is a secure channel to communicate the secret key, there will always be a risk that this channel is compromised by an attacker. [29]

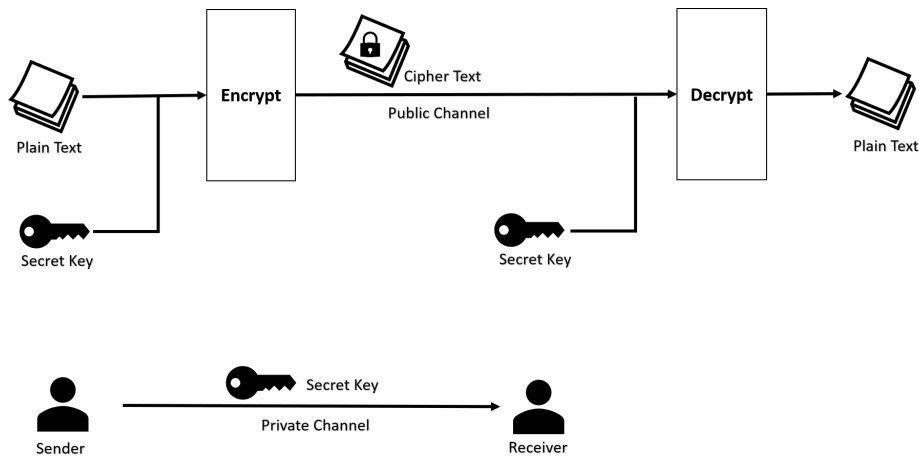


Figure 2.1 Symmetric Cryptography

Public Key Cryptography, also known as Asymmetric cryptography, was introduced to solve the problem of sharing the secret key [30]. In this model, the sender has a pair of keys called the public key and the private key; each party makes their public key accessible to all while keeping their private key for themselves. The brilliant feature is that the data encrypted with a public key can only be decrypted using its paired private key, and vice versa. Assume the previous scenario. In this model, the sender uses the public key of the receiver to encrypt the Plain-Text. As shown in Figure 2.2, in this model there is no need to communicate any keys between two parties, because the decryption is only possible by the receiver's private key which is supposed to be kept safely with its owner.

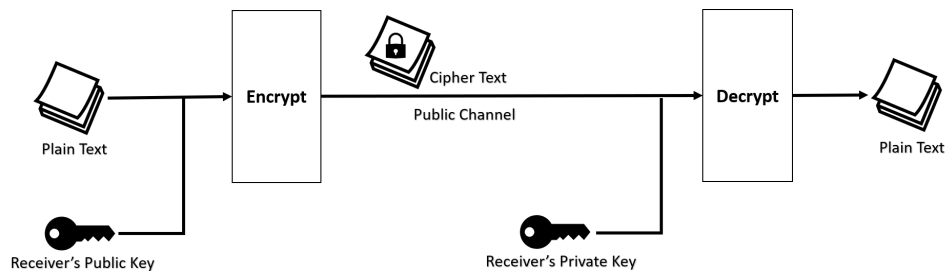


Figure 2.2 Asymmetric Cryptography

Digital Signature

One of the areas of usage of Public-key cryptography is its use in digital signatures. Digital signature authentication is a mechanism to verify the identity of a message sender [31]. Using public key and private key pair is a very secure and privacy-preserving method for digital

signature. Assume a sender wants to send a document to a group of people, and the receivers need to verify the identity of the sender. The sender hashes the document using his private key. Then he sends the hashed document, along with the original document to the verifiers. The only thing the verifiers need to do is to decrypt the hashed document using the sender's public key and compare it to the document they received from the sender. If they are equal they can approve that the document was signed using the sender's private key, as shown in Figure 2.3.

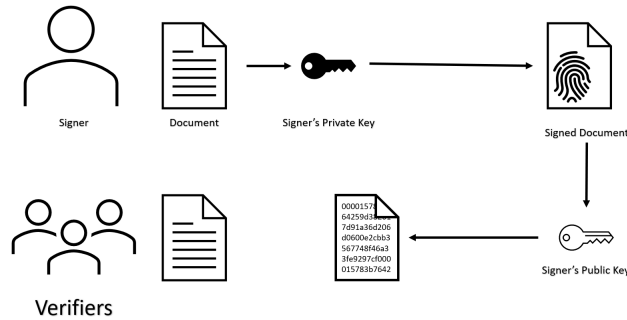


Figure 2.3 Digital Signature

2.1.2 Cryptographic Hash Functions

A hash function [32], [33] is a mathematical function that maps inputs from any type and size to outputs of the same type and size as shown in Figure 2.4. Hash functions are one of the concepts in which some of the most basic features of blockchain are built based on these functions; we will explain these usages of the hash function in blockchain in section 2.2.

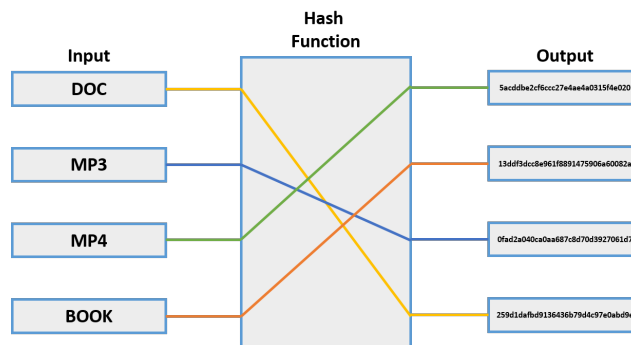


Figure 2.4 Cryptographic Hash Functions

A hash function can be considered as a encryption tool, which has some vital characteristics.

Assume that $f()$ is a hash function. $f()$ should be:

- Deterministic: for a specific input I_1 the hash function will always have the same output.
- Random output: The output of function is totally random. Assume there are two inputs x and y . There is no feasible way to predict the $f(y)$ based on $f(x)$.
- Irreversible: Hash functions are known to be one-way functions. Assume $y = f(x)$, where y is hash of x ; there is no computationally feasible way for someone having access to y to calculate x [34].
- Collision Resistant: Assume x and y as two different inputs to a hash function. The possibility of finding such x and y in which $f(x) = f(y)$ is near zero. [35]

Data Verification

One of the applications of Hash Functions is data verification. Assume that there is a piece of data D which should not be changed. Anybody can store the hash of data: $h_D = Hash(D)$ (Figure 2.5 (a)). Anytime later for checking originality of the current data content, they just need to recalculate the hash: $h'_D = Hash(D)$ and compare it with the original hash they have stored previously (Figure 2.5 (b)). Since the hash functions are deterministic, if D has not changed both hashes h_D and h'_D should be equal.

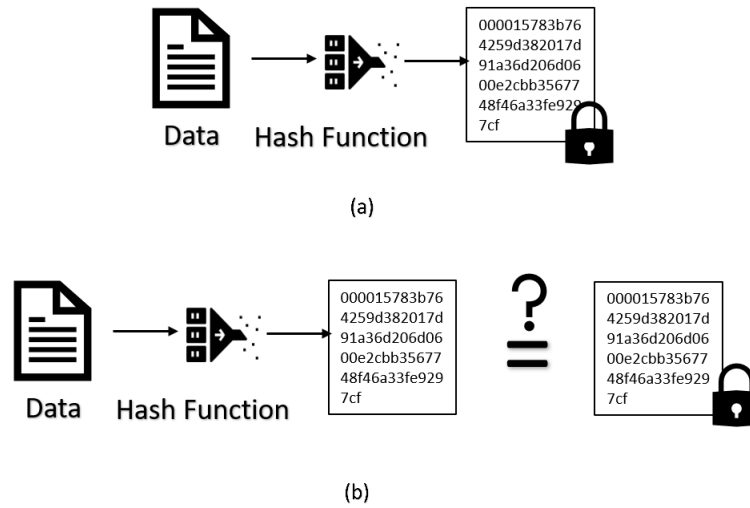


Figure 2.5 (a) Calculate the hash of a data and store it (b) recalculate the hash of data and compare it with the stored hash

2.2 Blockchain

Blockchain is best known for the fame of Bitcoin [36] as the first practical usage of this innovative technology. The idea of blockchain was so interesting that soon after the success of bitcoin many other platforms grew in the context of blockchain which were not limited to its usage as a cryptocurrency system. Ethereum is one of them that uses a Virtual Machine to run bytecodes, enabling Ethereum to be used by developers to implement their applications on Ethereum blockchain.

Blockchain, as its name implies, is a data structure based on a chain of blocks, which in combination with cryptography concepts and Peer-to-Peer network characteristics has gained the highest levels of security and immutability. A simple blockchain structure is represented in Figure 2.7.

2.2.1 Peer-to-Peer Networks

Blockchain is built on Peer-to-Peer networks. In this network participating nodes share their computational or storage resources with each other and communicate directly without intermediaries. All the nodes can be at the same time service provider and service requester. It means that in a P2P network all the nodes have the same responsibilities, in contrast to the Client-Server Networks which Server is responsible to respond to the requests of the

Clients [37]. The structure of a Client-Server and a P2P network is shown in Figure 2.6.

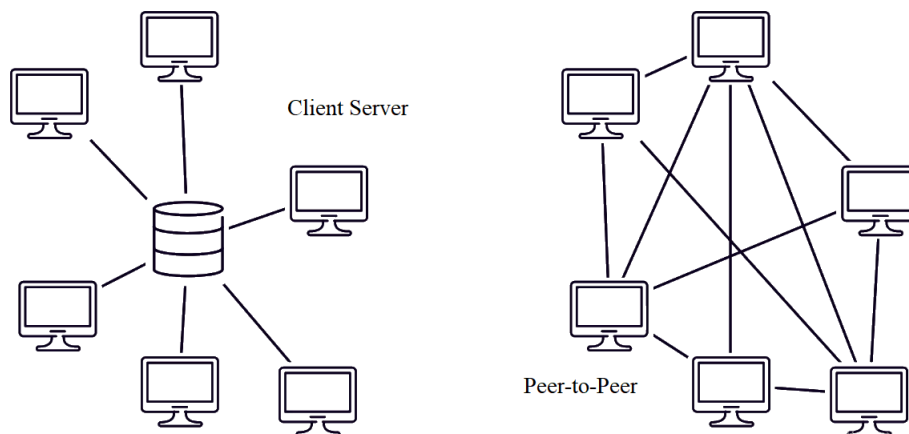


Figure 2.6 Client Server vs Peer-to-Peer network

In blockchain the decentralization is achieved by this P2P network structure. All the nodes in a blockchain are responsible to keep the last state of the system and update their state based on the received transactions in a new block.

2.2.2 Distributed Ledger Technology

A ledger in finance is recording the history of all financial events. Distributed Ledger Technology is a modern idea that owes its name to this traditional financial concept. A Distributed Ledger Technology is a P2P network in which all the nodes store an instance of the ledger. This was literally the basic concept on which bitcoin was built: a P2P network in which all the nodes keep a record of transactions; although with the advancements in the blockchain technology, the nodes are not just recording financial events of the system, but instead recording the last state of the blockchain. Different technologies are used to aim this: P2P Networks, Cryptography, and Consensus Mechanism [38].

2.2.3 Blockchain Structure

A simple blockchain structure is represented in Figure 2.7. We will explain the constituent elements of this structure. A blockchain is a chain of data structures called blocks that are linked cryptographically using hash pointers. Each block has a time stamp which is the time that the block is mined and verified. A block has two parts: block body and block header. The main parts of the block are described here.

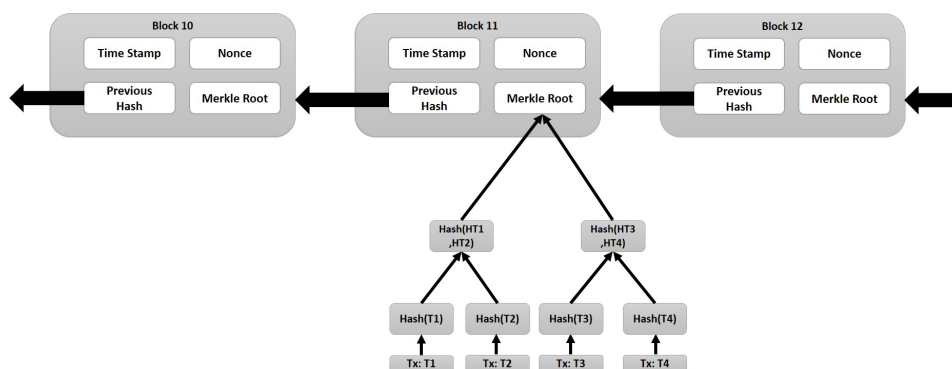


Figure 2.7 Blockchain structure

- **Blocks**

Blocks in blockchain are where the transactions are stored. Each block is created from a header and a body. The transactions are stored in the body, in a Merkle tree structure, and the root of this tree, which is the hash of the transactions is stored in the block header. We will talk about the Merkle tree later. Saving Merkle root in the header simplifies the verification and immutability of the data that are stored in the blocks.

- **Hash Pointer** Each block is linked to its previous block via a hash pointer, which is the hash of the previous block. Hash of a block is calculated from three pieces of data: the block transactions summarized in Merkle root, previous block hash, and a nonce that is random data used to solve the PoW puzzle.

- **Merkle Tree**

A Merkle tree is a binary tree data structure. The leaves in this tree are the hash of transactions. Each node stores the Hash of its child nodes. By calculating the hash of all nodes, the root will store the hash of all transactions contained in the leaves [39]. A Merkle tree that contains 4 transactions is shown in Figure 2.8.

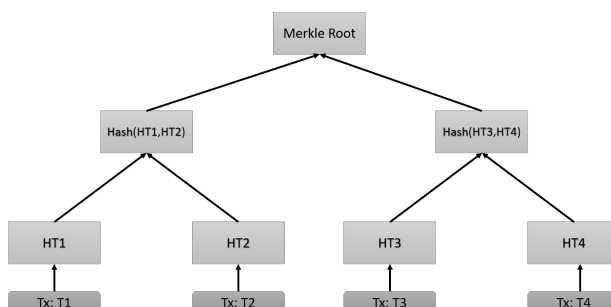


Figure 2.8 Merkle Tree structure

Merkle tree is an efficient data structure for both content authentication and transaction verification. Once the Merkle root is calculated and stored on a block, it is very easy to check if the contents of the block have not been tampered with. We can recalculate the hash root based on the block transactions and compare it to the hash root stored in the block header, and they should be the same. In addition, to verify the existence of a specific transaction in a block there is no need to calculate the hash of all transactions. It is possible with only $\log(n) + 1$ calculations, where n is the number of transactions. As you can see in 2.9, verification of existence of $T2$ is possible only with $Hash(T2)$, $Hash(HT1)$, and $Hash(HT3, HT4)$.

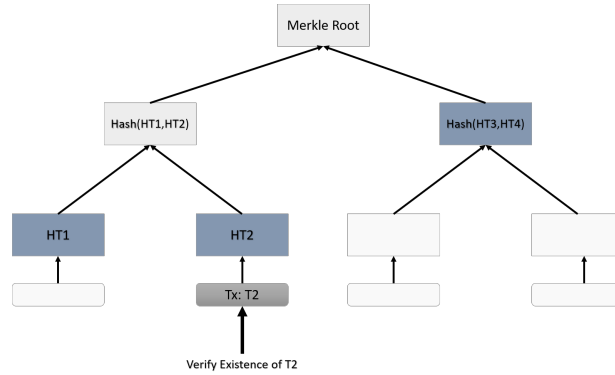


Figure 2.9 Transaction Verification in Merkle Tree

2.2.4 Consensus Mechanism

The main idea of blockchain is the decentralization of every possible operation including data storage, code execution, etc. in a Peer-to-Peer network. Everything in the blockchain system is happening via transactions. For example, a simple transaction in bitcoin is transferring 1 bitcoin from Alice's account to Bob's account. Transactions are broadcasted in the Peer-to-Peer network by users, and nodes are responsible to update the state of the system based on these transactions. An essential decision in the implementation of any decentralized framework is the way all nodes reach a consensus on the state of the system. Every node has its instance of the system, which should be updated once the majority of other nodes of the network reach a consensus on an update. In blockchain, this update is adding a new block including the transactions to the end of the chain of blocks. We will review some of the popular consensus mechanisms. Different consensus methods have been proposed after the advent of bitcoin which tried to make improvements in aspects of energy consumption and transaction throughput. [40]

- **Proof of Work** Bitcoin uses Proof-of-Work (PoW) consensus model. In PoW nodes which are known as miners try to solve a puzzle and the first one who wins by solving the puzzle sooner will be able to update the system's state via a new block and receive the mining reward. The puzzle in Bitcoin is a mathematical problem.

Once a new transaction is received in the network all the miners add it to their local memory called transaction pool. All the nodes create a Merkle tree of transactions in their mining pool which they have decided to place in the next block. Having the Merkle root and the hash of the previous block (Hash Pointer), the miner repeatedly changes the nonce value and calculates the hash of the block until achieving a hash which is accepted as the puzzle solution. In bitcoin, for example, the hash should start with a specific number of zeros. The miner who finds this hash will broadcast it to the network. All the nodes will verify if the claimed solution is valid and if it is valid they will add it to the blockchain. The miner who has solved the puzzle receives the mining reward specified in the protocol.

Solving a mathematical problem is an expensive task and guarantees the security of the system because only the nodes who have used their resources in the mining process are allowed to change the state of the blockchain.

- **Proof of Stake** In the proof of stake mechanism, the nodes who want to add a new block to the blockchain need to stake the native coin of the blockchain. All nodes are able to stake coins and become a miner. There is a simple formula that uses the age of staked coins to decide which node will add the next block to the blockchain. The longer a node has staked a coin, the more chance they will have to add the next block and receive the mining reward.

The PoS mechanism eliminates the high computational demand of PoW, so does not waste the resources of miners. Ethereum's initial consensus mechanism is PoW, but they have had the plan of switching to the PoS from the beginning. Eth 2.0 will make this move from PoW to PoS and is supposed to make a huge improvement in the Ethereum scalability.

- **Delegated Proof of Stake** In the Delegated Proof of Stake model, nodes need to stake the native coin of the blockchain like PoS, but they use their stake to vote to their witness to create the next block. The more votes nodes receive the higher chance they will have to create the next block.

2.2.5 Ethereum

Ethereum uses blockchain technology to make decentralized application development possible [41]. Ethereum has its own Virtual Machine called EVM, which can execute bytecodes. These low-level bytecodes are known as smart contracts. There are different high-level Turing complete languages like Solidity with compilers to translate their codes into EVM compatible bytecodes.

Some of the concepts of Ethereum are the same concepts we went through in the previous section. There are concepts which we will cover in this part [42].

I. Turing Completeness

Turing Completeness is a property of a state machine that is able to solve all the problems. A system is considered Turing Complete if it is able to simulate a Turing Complete State Machine. Ethereum Virtual Machine (EVM) has this property and so is an appropriate platform to compute any algorithms.

II. Account Types

In Ethereum there are two types of accounts:

- **Externally Owned Accounts** are the accounts managed by an external user and have an address, a balance, and a private key. These accounts are able to manage their balances and initiate transactions.
- **Contract Accounts** A contract account has an address and a balance like the EOA. In addition, it has its own smart contract code. They can send and receive Ether but they are not able to trigger a transaction by themselves. Once they are triggered they will be able to call other contracts.

III. Gas and Payment

All the operations on Ethereum are executed via transactions, and transactions vary in the complexity of their operations. In Ethereum, all of these operations are measured based on the effort is required for their execution. The gas unit is the measurement unit for these operating costs. Gas price is a scalar value in Wei (the smallest denomination of ether) which changes with the state of the network.

Ethereum uses gas units in conjunction with gas prices to protect the network from spammers since the transaction sender should pay the transaction cost which is calculated by multiplying the required gas units for the transaction with the gas price.

IV. Transaction

All the operations in Ethereum happen via transactions. As we mentioned before, transactions can be initiated by EOAs. To send a transaction an EOA needs to sign a message containing the details of their desired operations, and broadcast it to the P2P network of Ethereum. The transactions are verified and mined by the miners.

As we mentioned before, all the updates to a blockchain system's state occur via transactions. While in primitive blockchain systems like bitcoin the only data contained in a transaction was the cryptocurrency transfer, in Ethereum these transactions will consist of data and code, in addition to Ether transfer.

There are three types of transactions in Ethereum:

- The simplest transaction which like bitcoin transfers crypto between accounts.
- The Contract Creation transactions which deploy a contract on blockchain. These Contracts are stored on Ethereum and users will be able to execute their functions.
- The Contract Execution transactions which are triggered by users and contain functions and inputs parameters which are executed by EVM.

V. Ethereum Virtual Machine (EVM)

The smart contract deployment and execution is handled by Ethereum Virtual Machine. We can consider Ethereum as a world computer in which all the nodes of the network have their local instance of EVM. Every transaction which makes changes to the state of blockchain is executed on the EVM on all the participating nodes.

2.3 Zero-Knowledge Proofs: PLONK and SNARKs in Ethereum's Ecosystem

In this section, we provide a basic introduction to zero-knowledge proofs and how they impact the ethereum ecosystem.

Assume that there is a prover who wants to prove his knowledge of a specific piece of information to a verifier. Using the Zero-knowledge proof method gives this opportunity for the prover to prove their knowledge to the verifier without revealing any extra information. An intuitive example that is used to explain zero-knowledge proof is called the cave of Ali Baba. Assume that Alice and Bob are at the entrance of a cave, as pictured in Figure 2.10. This cave has two paths A and B , which are connected through a door. Bob claims he has a secret code that is the key to opening this door. In this example, Bob is the prover, and Alice is the verifier. Alice intends to buy the key but must ensure that Bob's claim is true before

buying. On the other hand, Bob intends not to reveal his code to Alice before selling. The solution is a Zero-knowledge method as follows. Suppose Bob enters the cave. Alice enters the cave after Bob and does not know which path Bob has chosen. In this situation, Alice asks Bob to leave the cave, for example, through path A, and Bob leaves the cave through path A. Here are two possibilities for Alice. There is a 50% chance that Bob has already entered path A, and another possibility is that Bob was in path B and held the key and entered path A through the door. This possibility is not enough to convince Alice. Alice has to repeat this experiment many times. Each time Bob succeeds in this experiment, the chances of Bob accidentally entering the cave from Alice's requested path are reduced to the point where Alice is convinced that Bob has the key.

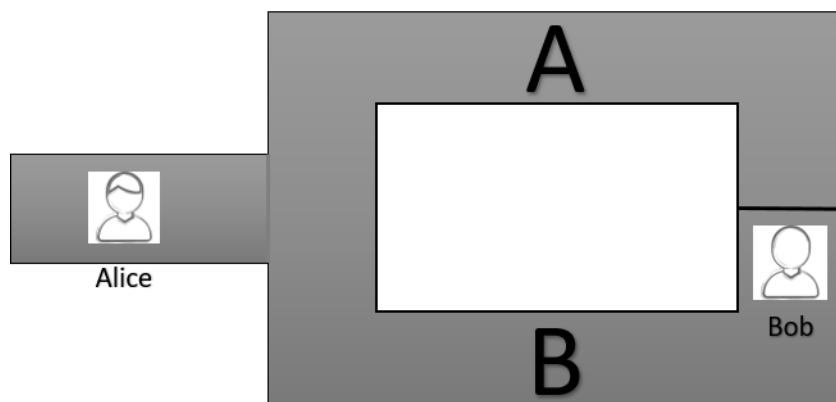


Figure 2.10 Ali Baba Cave

This is the basic idea on which interactive zero-knowledge proof protocols like Schnorr protocol [43] and Pedersen protocol [44] are built on it. This is called interactive since the prover and verifier need to interact with each other to convince the verifier about the knowledge of the prover.

Another type of Zero-Knowledge Proof protocol is non-interactive ones. As the name suggests, the prover and verifier do not need to interact and instead use cryptographic methods [45].

A visual representation of the evolution and categorization of zero-knowledge proofs is depicted in Figure 2.11, illustrating the various zk-SNARK and zk-STARK systems along with their notable implementations.

Halo2 and PLONK represent two advanced but distinct approaches within the landscape of

plonkish zk-SNARKs, each categorized based on their core cryptographic techniques and use cases. PLONK utilizes a universal and updatable structured reference string (SRS), making it a versatile choice for a wide range of applications without the need for specific circuit recompilations for different proofs. This universality reduces the overhead associated with multiple setups and is particularly beneficial in environments where the same circuit is reused extensively. In contrast, Halo2, designed to support recursive proof compositions [46], does not require a trusted setup, addressing one of the significant concerns in the cryptographic community regarding the potential vulnerabilities associated with the initialization phase in SNARKs. Instead, Halo2 enables the construction of proofs that can verify other proofs, a feature that drastically scales the verification of complex and nested transactions common in blockchain technologies. While PLONK fits into the category of non-recursive zk-SNARKs ideal for static or less frequently changing computational environments, Halo2 falls into a specialized niche of recursive SNARKs, catering to dynamic systems requiring ongoing proof validations without the scalability constraints imposed by traditional SNARKs.

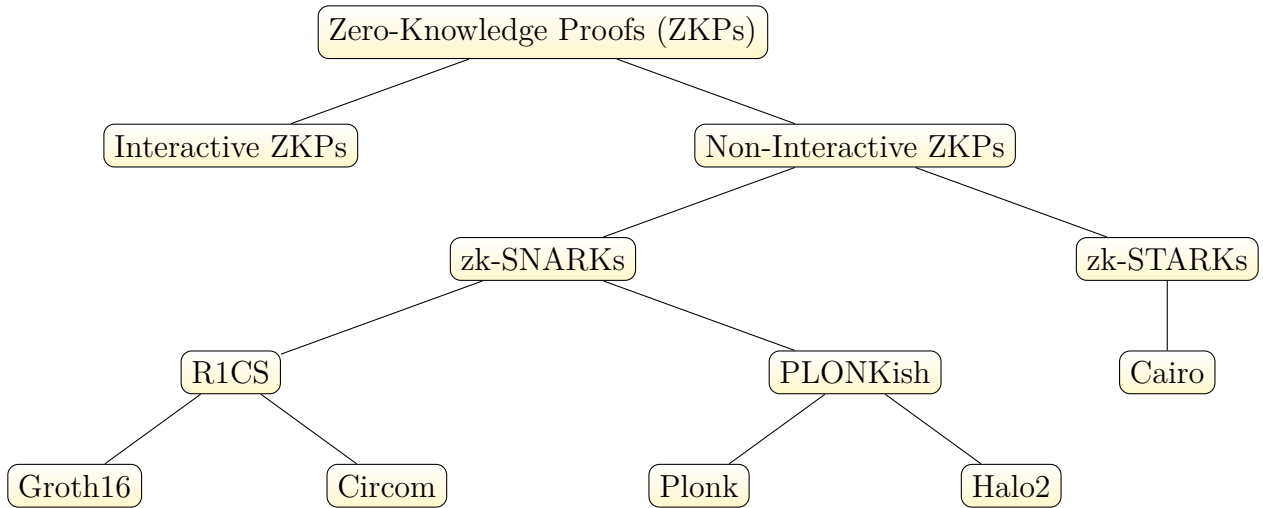


Figure 2.11 Overview of Popular Zero-Knowledge Proofs (ZKPs), showcasing key interactive, non-interactive, zk-SNARK, and zk-STARK frameworks

2.3.1 Fundamentals of SNARKs

At the heart of Zero-Knowledge Proofs (ZKPs) lies the capability of one party to prove the truthfulness of a statement without revealing any information beyond the fact that the statement is true. Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge, or ZK-SNARKs, stand out for their efficient proof sizes and swift verification processes, which scale sublinearly with the complexity of the statement being verified. Furthermore, ZK-

SNARKs enable a non-interactive exchange between the prover and the verifier, ensuring security even against adversaries with bounded computational capabilities.

ZK-SNARKs are particularly adept at addressing the circuit satisfiability problem, representing statements as arithmetic circuits. This capability facilitates the proof of a fixed NP relationship R , allowing a prover to convince a verifier of knowing a witness w for a public input x , such that $(x, w) \in R$. The introduction of pre-processing SNARKs enriches this discussion by incorporating a crucial setup phase that encodes the relationship into a succinct format, optimizing both the proving and verification processes.

The SNARK framework is underpinned by three pivotal algorithms: The first step, is the initial setup for the proof and verification process, where an algorithm initiates the system with public parameters, generating the proving key and the verification key. With the proving key, public input, and witness, the prover generates the proof. Then, the verifier utilizes the verification key, public input, and proof to ascertain the proof's validity.

Security guarantees inherent in ZK-SNARKs include:

- **Knowledge Soundness**

Knowledge soundness is a critical security property ensuring that a dishonest prover cannot convince the verifier of the truth of a false statement. For ZK-SNARKs, this property is formally defined as follows:

Definition: A zero-knowledge proof system possesses knowledge soundness if it is computationally infeasible for any prover, without a valid witness w , to produce a proof π that convinces the verifier of the truth of a false statement s .

Implication: This property ensures the integrity of the proof system by preventing malicious actors from manipulating or fabricating proofs, thus undermining the system's security.

- **Completeness**

Completeness in a zero-knowledge proof system guarantees that an honest prover, who possesses a valid witness, can always convince the verifier of the truth of a legitimate statement.

Definition: A zero-knowledge proof system is complete if, for every true statement s and valid witness w , there exists a proof π such that the verification algorithm $V(s, \pi) = 1$.

Implication: This property affirms the reliability of the proof system, ensuring that

valid assertions are always verifiable, thereby fostering trust in the system’s functionality.

- **Zero-Knowledge**

The zero-knowledge property ensures that no additional information about the witness w is revealed to the verifier apart from the fact that the statement s is true.

Definition: A zero-knowledge proof system upholds the zero-knowledge property if, for every proof π that the verifier accepts, no information other than the validity of the statement s can be inferred about the witness w .

Implication: This property safeguards the privacy of the underlying information, ensuring that sensitive data remains confidential even in the process of verification.

Together, these properties make ZK-SNARKs a powerful tool for applications in blockchain and beyond. By ensuring that proofs are sound, and complete, and reveal no additional information, ZK-SNARKs provide a secure and efficient method for verifying transactions and other data interactions in decentralized systems without compromising on privacy or security.

For those seeking an in-depth understanding of ZKPs and SNARKs from both a theoretical and practical perspective, further exploration is encouraged through foundational works such as those by Kilian [47] and others who have significantly contributed to the field.

The development of SNARKs has led to a variety of implementations, each tailored to specific requirements and use cases, including Pinocchio [48], SONIC [49], Groth16 [5], PLONK [2], and Halo [6]. These systems underscore the practical applicability and versatility of SNARKs in enhancing digital security and privacy.

PLONK’s framework and its innovations have also influenced the development of halo2, an advancement in SNARK technology. halo2 aims to further optimize protocols like PLONK, especially in terms of scalability and efficiency, and is a focus of subsequent chapters in this thesis.

2.3.2 Arithmetization in Zero-Knowledge Proofs

Arithmetization is the process of converting logical and algorithmic statements into algebraic equations that form the basis of zero-knowledge proofs. This section details how arithmetization operates within the Halo2 framework and its reliance on tables for managing constraints and optimizing proof verification. Arithmetization transforms complex computations into

a set of polynomial equations. These transformations are essential for enabling the cryptographic systems to efficiently verify the computations without executing them. The process involves several key steps:

1. **Encoding Logic and Data:** Initially, the data and logical operations involved in a computation are encoded as elements of a finite field. This encoding translates each operation and variable into algebraic expressions.
2. **Forming Constraint Systems:** The entire computation is broken down into a system of constraints that describe the relationships between the variables. These constraints ensure that any solution to the system corresponds to a valid execution of the original computation.
3. **Generating Polynomial Equations:** The constraints are then converted into polynomial equations. The goal is to construct a set of polynomials whose solutions correspond to the solutions of the original constraint system.

Tables

Tables in arithmetization are akin to databases or spreadsheets where data is organized systematically for easy access and manipulation. These tables contain different types of columns that store specific data types:

- **Instance Columns:** These columns hold public inputs to the circuit. They are visible to both the prover and the verifier.
- **Advice Columns:** Advice columns store private inputs or intermediate values computed during the execution of a circuit. These are generated by the prover and are crucial for constructing the proof but do not need to be disclosed to the verifier directly.
- **Fixed Columns:** Fixed columns contain constants or pre-computed values that are known beforehand and used throughout the computation. These are akin to parameters or configuration settings for the circuit.

Gates

Gates in the context of zero-knowledge proofs are logical units that enforce specific arithmetic or logical conditions. Each gate can be thought of as a constraint or a rule that inputs must satisfy:

- Gates define the relationship between inputs and outputs in a circuit, effectively dictating how inputs are transformed into outputs.
- Each gate in a circuit corresponds to a row in the arithmetization table.

Cells

Cells represent individual data points within the tables. Each cell is associated with a particular column and a specific gate:

- Cells hold values that are either inputs to the gates, outputs from the gates, or intermediate values needed for the computation.
- The value in each cell must satisfy the constraints imposed by its corresponding gate to ensure the circuit as a whole functions correctly.

2.3.3 Utilization of Tables in Halo2

In Halo2, tables are used to orchestrate the complex relationships and dependencies involved in zero-knowledge proofs:

- **Constraint Evaluation:** Halo2 utilizes tables to evaluate the constraints associated with each gate efficiently. This is done by organizing the arithmetic relationships in a manner that aligns with the polynomial commitments.
- **Look-Up Tables:** Halo2 often incorporates look-up tables to manage complex functions that are difficult to represent directly through simple arithmetic gates. These tables help in verifying that certain conditions or relationships hold without explicitly computing them each time.

Having explored the intricate components of arithmetization within zero-knowledge proof systems such as Halo2 and PLONK, we now shift our focus towards their practical applications in enhancing blockchain technologies, specifically within the Ethereum ecosystem. The scalability of Ethereum is a pivotal area where the advancements in cryptographic proofs, particularly those facilitated by sophisticated arithmetization techniques, play a crucial role. As Ethereum grows in usage and complexity, the need for efficient, scalable solutions becomes increasingly critical. In the following section, we will explore how the principles and technologies discussed contribute to addressing Ethereum's scalability challenges, highlighting the integration of zero-knowledge proofs as a core component of scalable blockchain architectures.

2.4 Ethereum Scalability

Ethereum has been the most favoured ecosystem for developing smart contracts recently; although, even the founders of Ethereum are aware of the scalability issues developers are facing while developing smart contracts. Transaction throughput can be considered as a parameter for measuring the scalability of blockchain systems. In Ethereum with the increasing number of users not only the finality of transactions and as a result, the throughput decreases, but also the cost of executing a transaction on mainnet rises. Due to these problems, there are two solutions in the Ethereum community developing at the same time: Eth 2.0 and Layer 2 solutions. [50].

2.4.1 Ethereum 2.0

Ethereum 2.0 proposes combining two technologies to increase scalability: [51]

- **Proof of Stake** The Proof of Stake as is described in 2.2 is more efficient than PoW. Ethereum switches totally to PoS in Eth 2.0.
- **Sharding** is a model of state management in which the state of the blockchain is distributed among interrelated blockchains called shards [52]. One shard is responsible for coordinating all other shards. This can reduce the workload on nodes since they are engaged with a subset of the network instead of the whole network.

2.4.2 Layer 2

Another actor engaging in blockchain ecosystem called layer 2 is not doing less to make Ethereum scalable. Layer 2 is a solution that allows handling operations off-chain while taking advantage of the security of mainnet. We will explain one of the most popular ones called Rollups [13].

Rollups

Rollups are the idea of separating execution from data to reduce the cost of running applications on blockchain. The idea is that there is no need to do all executions on the main chain, called layer 1; executions can be performed outside the main chain, on layer 2, and only the data is stored on the main chain (on layer 1) as proof of transactions. We will explain two popular Rollup methods: Zero-knowledge Rollups (ZK rollups) and Optimistic Rollups [53].

- **Zero knowledge Rollups**

Zero-knowledge Rollups also known as ZK-Rollups, as their name suggests are based on Zero-Knowledge Proofs. To explain how ZK-Rollups works, let us first describe the concept of Zero-knowledge Proof. The ZK-SNARK, one of the non-interactive Zero-Knowledge Proof protocols, is the protocol used in ZK-Rollups.

With understanding how Zero-Knowledge Proof works, we can now return to the ZK-Rollups discussion. In this method, there is a smart contract on the Ethereum mainnet that maintains the status. In a Layer 2 network, there are two types of actors: those that send transactions to the network and those that collect these transactions to execute them and send to the Layer 1 network, called aggregators. In this method, all calculations are performed on layer 2 by aggregators. Aggregators compress a lot of transactions into one transaction. Then all the transaction data along with the new state are sent to Layer 1 in one condensed transaction. Since the transactions are not executed on mainnet, this transaction consists of a ZK-SNARK that proves the validity of executed transactions and the new state.

There are two weak points about the ZK-Rollups: firstly, they are mostly used for transactions containing simple token transfer rather than those containing smart contracts; secondly, calculating ZK-SNARK is a computationally extensive task.

- **Optimistic Rollups**

Optimistic Rollups are built on top of Ethereum layer 1 blockchain [54]. They have a virtual machine called OVM which executes any kind of transactions that EVM can. So far, the concept of executing transactions off-chain (on layer 2) is similar to ZK-Rollups. In the Optimistic Rollup method, the validity of transactions are not verified and all of them are sent to the smart contract on mainnet, but they are not assumed finalized for a period of time, and in this period the nodes who find out about a fraud, called fraud proof, will receive rewards. This is the reason why they are called optimistic. The fraud-proof is what guarantees the validity of transactions in this method, while in ZK-Rollups the ZK-SNARKs prove the correctness of transactions.

2.4.3 Zero-Knowledge Ethereum Virtual Machines (zkEVMs): Enhancing Privacy and Scalability in Ethereum

Introduction to zkEVMs

Zero-Knowledge Ethereum Virtual Machines (zkEVMs) represent a significant technological advancement in the Ethereum ecosystem. They integrate zero-knowledge proof concepts with

Ethereum’s virtual machine architecture.

The Role of zkEVMs in Enhancing Ethereum’s Scalability through zkRollups

Zero-Knowledge Ethereum Virtual Machines (zkEVMs) significantly contribute to the scalability of the Ethereum network, especially through their role in zkRollups. As a layer 2 scaling solution, zkRollups utilize zkEVMs to process many Ethereum transactions off-chain efficiently. This is achieved by batching and executing these transactions within the zkEVM environment. Subsequently, zkEVMs generate a succinct zero-knowledge proof representing the validity of the entire batch of transactions. This proof is then submitted to the Ethereum mainnet and verified. This approach dramatically reduces the on-chain data and computational load, enhancing the Ethereum network’s throughput. The integration of zkEVMs in zkRollups addresses a major scalability challenge, enabling a higher transaction throughput and improved overall efficiency of the Ethereum blockchain, making it a pivotal advancement in Ethereum’s ongoing scalability efforts.

2.5 Satisfiability Modulo Theories (SMT) Solvers

Satisfiability Modulo Theories (SMT) solvers are advanced computational tools used to determine the satisfiability of logical formulas over one or more theories. These solvers enhance the capabilities of Boolean satisfiability solvers (SAT solvers) by enabling the handling of formulas with complex data types such as integers, real numbers, and arrays. Such capabilities are crucial in fields like software verification, optimization problems, and cryptographic systems, including zero-knowledge proofs, which are over finite fields.

SMT solvers are particularly useful in cryptographic contexts for checking potential vulnerabilities in circuits, such as those arising from under-constrained or over-constrained conditions. A key advantage of these solvers is their ability to model all the constraints within a circuit into an equivalent set of constraints that are both compatible and solvable by the solver.

The choice of the CVC5 solver for analyzing zero-knowledge proofs that utilize finite fields is primarily driven by its unique support for Finite Field Theory among SMT solvers. This capability makes CVC5 ideally suited for the complex requirements of cryptographic proof verification, where precise modelling and solving of field-specific constraints are necessary.

Implementation of zkEVMs Using halo2 in zkRollups Several zkRollup solutions within the Ethereum ecosystem leverage the advanced capabilities of halo2 for implementing

their Zero-Knowledge Ethereum Virtual Machines (zkEVMs). halo2, as a state-of-the-art zero-knowledge proof system, offers significant improvements in terms of efficiency and scalability, making it an ideal choice for zkRollups. By integrating halo2, these zkRollups can execute Ethereum transactions more efficiently, benefiting from halo2's optimized proof generation and verification processes. This integration enhances the scalability of these rollups and aligns with Ethereum's broader objectives of increasing transaction throughput and network efficiency. The use of halo2 in zkEVMs exemplifies the cutting-edge advancements in cryptographic proofs, underlining their growing importance in developing scalable blockchain technologies.

The Importance of Bug-Free halo2 Circuit Implementation in Ethereum's Ecosystem Implementing bug-free halo2 circuits holds critical importance in the Ethereum ecosystem, particularly because these circuits form the backbone of several zkRollup solutions. halo2, renowned for its advanced zero-knowledge proof capabilities, plays a pivotal role in ensuring the security, privacy, and scalability of transactions on the Ethereum network. Any bugs or vulnerabilities in these circuits could lead to significant security risks, including the potential for malicious transaction verifications. This makes the task of ensuring the robustness and reliability of halo2 circuits paramount. Consequently, developing tools to statically or formally verify these circuits becomes essential. Such tools are crucial in preemptively identifying and rectifying potential weaknesses or bugs in these circuits, thereby fortifying the Ethereum ecosystem against security threats. The reliability and security of halo2 circuits, ensured through rigorous analysis, directly contribute to the overall stability and trust in Ethereum's blockchain infrastructure, especially as it continues to scale and incorporate more complex smart contract operations.

2.6 Access Control

The main goal of an access control system is defining policies to decide which user has access to which objects. The object is a general term that can refer to a wide range of concepts. For example, in an operating system, the objects can be system files or directories. In our study, we define objects as the functions of a smart contract, and the access control system manages how users access system functions. Different access control Systems suggest different methods to define these policies.

In this part we discuss two popular access control models, role-based access control and attribute-based access control, and their strengths and weaknesses.

2.6.1 Role-Based Access Control: RBAC

Role-based access control model [55] defines a category of roles. The roles are assigned to users, , and policies are defined for these roles, as you can see in Figure 2.12. This Model will reduce complexity if there is not a lot of variety or changes in the roles. In fact, this model is appropriate if there are a limited number of roles and the changes are mostly in the users' roles. Otherwise, there is a risk of role explosion; like a system that the admin has to define a specific role for each user.

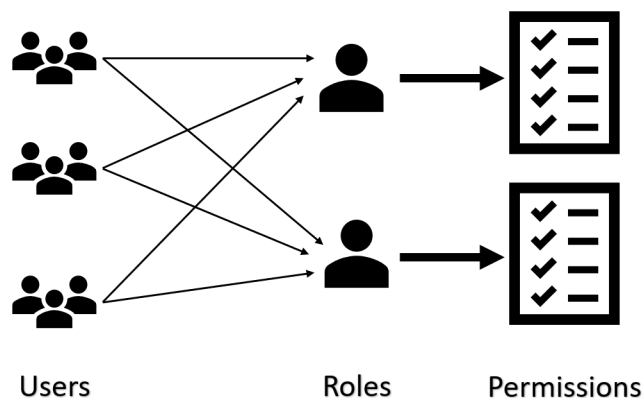


Figure 2.12 Role-Based Access Control Model

RBAC provides simplicity and ease of administration [56]. However, the role definition can become a crucial task, and if the roles are not identified properly there is a risk of role explosion.

2.6.2 Attribute-Based Access Control: ABAC

In the attribute-based access control model, attributes are assigned to each subject, object, and operation request by the authorities; then, the access control policies are written and controlled based on these attributes [57]. The ABAC model is shown in Figure 2.13.

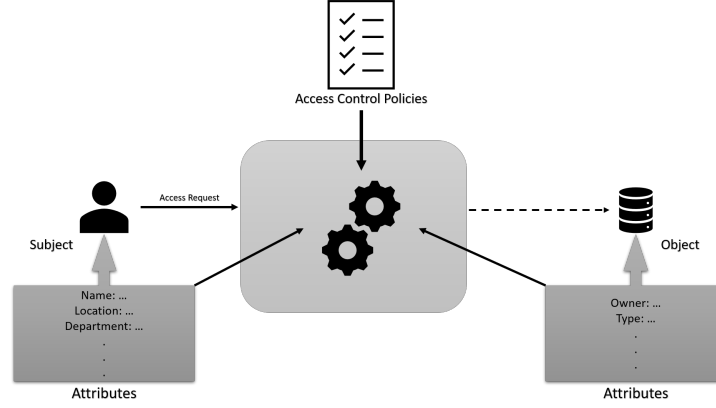


Figure 2.13 Attribute-Based Access Control Model

Flexibility is something that is gained in the ABAC models since there is the possibility of fine-grained Access Control without limitation. Although, it makes the implementation, management, and auditing of the policies very complex.

2.6.3 RBAC vs ABAC

Theoretically, ABAC can be considered as a superset of RBAC. It means that a role can be considered as an attribute, like other attributes defined in the ABAC model. So in the first look, it seems that ABAC has all the qualities of RBAC with the opportunity to have a more powerful fine-grained access control. But the fact about ABAC is that each attribute is just a string, while roles are categorical data, making them feasible for managerial tasks. Think about making queries on which users with role R has executed a specific function in both scenarios: in RBAC you just need to look for users assigned to the role R , within a group of known roles, while in ABAC there is no feasible way except looking through all the attributes, which are plain strings, to check if there is role attribute for each user. There are solutions which does not remove the role as the main characteristic of users, while enables flexibility of using attributes as there is in ABAC.

2.6.4 Combining RBAC and ABAC: Role-Centric Access Control

Authors in [56] proposed different possible combinations of RBAC and ABAC to use the best of both. Different combinations trade-off between different pros and cons of the two methods. As mentioned in the previous part, and according to the purpose of our study which is addressing access control issues to smart contracts functionality, we decided to use the role-centric access control model. Authors of [58] provided a formal model of the role-

centric access control model. In this model, the roles are defined and assigned to users. The policies are defined for these roles, and it is possible to restrict each role's permission level by defining some attributes for the users and adding some attribute-based rules to the roles' permission policy.

It is important to emphasize that the core innovation of this research is not in the creation of the role-centric access control model itself but rather in its novel adaptation to the Ethereum smart contracts.

To clarify, assume a simple smart contract which implements lending and borrowing of assets. So there is a Rent function and a Borrow function which are the objects we need to define policies to control who has access to execute them.

Based on this scenario the system admin defines two LENDER and BORROWER roles. A simple RBAC policy can be defined which allows the LENDER to execute Lend function, and BORROWER to execute the Borrow function. Assume that the contract admin decides that users with role BORROWER who work in company *C1* should not be able to borrow assets. In this case, these rules can be defined using an attribute COMPANY, i.e., if the attribute COMPANY of a user equals *C1*, access is denied. The access control system first controls a user's role to check if his role have access to the requested resource, and if a user is approved based on their role, then ABAC rules are controlled.

Consider that defining attributes is optional, and while the pure RBAC model is enough to answer the system requirements, this model will not get engaged with the complexity of defining attribute-based rules. For example, in mentioned sample contract, the admin is required to add the COMPANY attribute and set its value for users, at least for those who are subject to the restrictions.

2.7 Conclusion

This chapter has meticulously explored the foundational elements that underpin the core aims of this thesis, which focuses on enhancing the security of blockchain technologies, particularly Ethereum. By delving deeply into blockchain technology and its most prominent application through Ethereum, we set a solid groundwork for understanding the specific security challenges and scalability issues addressed in this research.

We have also provided an extensive introduction to zero-knowledge proofs (ZKPs), particularly zk-SNARKs, and their integration into Ethereum's ecosystem. This discussion is crucial as it underlines the importance of ZKP security in bolstering the overall security framework of blockchain technologies. Given the reliance of Ethereum on ZKPs for scalability solutions

such as zkRollups, understanding the intricacies of these proofs and their vulnerabilities is essential for developing more secure and efficient systems.

In addition to ZKPs, we briefly discussed the role of Satisfiability Modulo Theories (SMT) solvers, which are instrumental in our Halo2 circuit analyzer. These solvers enable us to verify the correctness and security of halo2 circuits.

Parallel to these cryptographic and verification tools, we examined access control mechanisms, laying the groundwork for understanding the novel smart contract access control management framework developed in this thesis. By outlining the principles of role-based (RBAC) and attribute-based (ABAC) access control systems, we provide the necessary context for appreciating how these models can be applied to manage access in decentralized environments.

Collectively, the background provided here serves not just to inform but also to contextualize the subsequent contributions of this thesis. It emphasizes how each aspect—from blockchain fundamentals to advanced cryptographic techniques—contributes to addressing the multifaceted challenges faced by Ethereum. The following chapters will build on this foundation, presenting methodologies and tools developed through this research to significantly advance the security of Ethereum ecosystem.

CHAPTER 3 RELATED WORK AND LITERATURE REVIEW

3.1 Tools for Analyzing Proof Systems

Circomspect is an open-source static analyzer and linter specifically designed for the Circom programming language, which is widely used in creating circuits for zero-knowledge proofs. The tool performs syntactic pattern matching on the abstract syntax tree of Circom source code to detect potential issues. In Circom, the correct usage of assignment is typically denoted by `<==`. A notable feature of Circomspect is its treatment of the assignment operator (`<-`) within Circom programs. It flags all occurrences of this operator as potential bugs, which leads to a high rate of false positives. However, this pattern-matching approach may not only generate many false positives but also miss genuine bugs. This approach provides developers with an initial layer of scrutiny, though it has limitations in precision and completeness.

Determinism in zero-knowledge proofs (ZKPs) is pivotal, enabling one to assert knowledge of a solution to a system of equations while maintaining secrecy about some or all components of the solution. These equations, denoted as $\Phi(X, Y)$, should ideally function such that the input X reliably produces the output Y . To validate this, a satisfiability query is employed, written as $\Phi(X, Y) \wedge \Phi(X', Y') \wedge X = X' \wedge Y \neq Y'$, indicating nondeterminism if satisfiable [59]. The significance of determinism extends beyond theoretical interest, as its violation has led to notable security flaws, such as in the Tornado Cash incident, and contributes to a significant proportion of the vulnerabilities tracked in ZK applications [60].

Ecne, similar to our SMT-based analyzer, is designed to ensure this kind of determinism. Ecne checks the intermediate representations of proof systems that use Rank-1 Constraint System (R1CS), verifying the soundness of R1CS equations to ensure they uniquely determine outputs from given inputs. The goal is to validate that R1CS constraints not only have unique solutions but also correspond to formally specified mathematical functions. Furthermore, Ecne attempts to verify that the input to a zero-knowledge proof (ZKP) encodes a deterministic computation, although it does not encompass any notion of a specification of the intended behavior of the computation.

Picus [16] marks a significant advancement in the static analysis of Zero-Knowledge Proof (ZKP) circuits by being the first tool to utilize SMT solvers for investigating underconstrained cases. While it adeptly identifies and flags underconstrained signals in ZKP circuits, it still grapples with the challenge of overconstrained scenarios and the limitations posed by the computational intensity of large circuits, which often exceed the capabilities of SMT solvers.

within reasonable timeframes. Picus supports a variety of circuit types, including those from Circom, R1CS, and gnark, requiring specific setups such as the circomlib and gnark library. The tool’s functionality extends to offering configurable settings for in-depth analysis and providing outputs that classify circuits as “safe,” “unsafe,” or “unknown,” thereby assisting researchers and developers in assessing the integrity and security of ZKP circuits.

Ecne and Picus are both tools for checking intermediate representations of proof systems that use a R1CS intermediate representation. These are incompatible with the Rust files used in the Halo2 proof system without additional effort. For instance, R1CS only supports quadratic polynomials, while Halo2 polynomials may have arbitrary degrees.

Vella and Alt [61] present a similar tool to check R1CS circuits, which uses SMT solvers to reach conclusions over large finite fields. Their research focuses on exploring algebraic techniques with the ambition of developing a hybrid algebraic-SMT decision procedure. This innovative approach aims to compute the satisfiability of a newly proposed theory of polynomials over large prime fields.

QED² system by Pailoor et al. [17] is described for automated detection of under-constrained systems in R1CS using the same cvc5 solver. Their approach performs *semantic reasoning* over R1CS equations to determine if there is a unique solution, and may be more general than our approach. Their approach may apply to Halo2 circuits with some modification, though they point out that it may not benefit from their uniqueness constraint propagation.

ZKAP [62] introduces a static analysis framework for circom that leverages a circuit dependence graph (CDG), an abstraction that encapsulates critical circuit characteristics and facilitates the expression of semantic vulnerability patterns through CDG-based queries. The framework includes nine distinct detectors including under-constrained circuits.

SNARKProbe [63] is an automated security analysis framework for zkSNARKs. This framework is designed to scrutinize R1CS-based libraries and applications, identifying a range of issues from edge case crashes to cryptographic errors and discrepancies with protocol specifications. *SNARKProbe* utilizes diverse analysis methods, including fuzzing and SMT solvers, to ensure robustness.

3.1.1 Domain-Specific Languages for Zero-Knowledge Proofs

As the field of zero-knowledge proofs evolves, several domain-specific languages (DSLs) have been developed to address the unique challenges and requirements of designing and implementing ZKP systems. Domain-specific languages (DSLs) such as Coda and Leo are advancing the capabilities for formal verification of circuits, offering developers powerful tools to

create more secure zero-knowledge applications.

- **Orbis** [64] offers a specification language for zk-SNARK programming, focusing on the creation of zero-knowledge proofs without addressing the verification of circuit correctness. This specialization allows developers to write Halo2 circuits more efficiently, although it does not provide tools for verifying the correctness of these circuits.
- **Noir** [65], developed by Aztec Labs, is tailored specifically for SNARK proving systems. It allows for the creation of complex zero-knowledge programs using an intuitive syntax accessible even to those without a deep understanding of the underlying mathematics. Noir works by compiling programs into an adaptable intermediate language called ACIR, which can then be transformed to fit various proving backends. This backend-agnostic approach ensures that Noir can be used across a range of applications and setups, making it a versatile tool in the ZKP landscape.
- **NoName** [66] is designed for the Kimchi zero-knowledge proof system and draws heavily from Rust's features, integrating some Golang concepts. It stands out for its robust debugging capabilities, providing detailed insights into how programs are translated into gate-level representations. This feature is crucial for developers who need to ensure the accuracy and efficiency of their zero-knowledge proofs.
- **Leo** [67], specifically designed for zero-knowledge applications, focuses on providing robust guarantees for custom-written zero-knowledge languages. Leo achieves two crucial properties: formal verification of applications against their high-level specifications, and the ability for applications to be succinctly verified by anyone, regardless of application size.
- **CODA** [68], a statically-typed language, enhances the development of zero-knowledge applications by enabling formal specification and static checking of properties through an advanced refinement type system. One of the primary challenges in the formal verification of ZK applications is the requirement to reason about polynomial equations over large prime fields, a task that stretches the capabilities of automated theorem provers.

These languages contribute significantly to the development and implementation of zero-knowledge proofs by providing tailored solutions that address the specific needs of cryptographic proof systems. Their development represents a significant step forward in the practical application of zero-knowledge proofs, especially in enhancing the security and efficiency of blockchain technologies.

3.1.2 Comprehensive Comparison of Zero-Knowledge Proof Analysis Tools

This section presents a detailed comparison of various tools developed for analyzing zero-knowledge proof systems. The comparison focuses on their functionalities, targeted proof systems, methodologies, and any unique features they offer.

Tool	Functionality	Proof Systems	Methodology
Circomspect	Linting/Static Analysis	Circom	Syntactic Analysis
Ecne	Determinism Checks	R1CS	SMT Solver
Picus	Underconstraint Analysis	R1CS, Circom	SMT Solver
ZKAP	Security Framework	Circom	Circuit Dependence Graph
SNARKProbe	Security Analysis	R1CS	Fuzzing, SMT
NoName	High-Level Programming	Kimchi	Language Design

Table 3.1 Overview of Tools for Zero-Knowledge Proof Analysis

The table illustrates a spectrum of tools each catering to different aspects of ZKP development and analysis. Tools like **Circomspect** and **Picus** focus on the syntactic and structural integrity of circuits, respectively, while **Noir** and **NoName** provide frameworks for developing ZKPs with higher-level programming abstractions. **Orbis** offers a unique approach by providing a specification language tailored for Halo2, enhancing the verifiability and correctness of implementations. This variety reflects the diverse needs and challenges in the field of zero-knowledge proofs, indicating the dynamic and multifaceted nature of research and development in this area.

To our knowledge, no automated tools currently exist to verify the security aspects of applications built using PLONKish arithmetizations. Addressing this significant gap, our research proposes a novel tool specifically designed for analyzing one of the most popular PLONKish arithmetizations, Halo2 circuits. This tool utilizes the CVC5 SMT solver coupled with abstract interpretation techniques to effectively identify both under and over-constrained circuits. Our approach not only fills a crucial void in the cryptographic verification landscape but also enhances the reliability and security of implementations, which are a building block of many blockchain scalability solutions.

3.2 Blockchain-Based Access Control Management for On-Chain DApps

The existing literature on access control solutions for smart contracts contains both practical libraries and theoretical articles. The integration of these two aspects is essential since they frequently intersect, imposing a collective impact on the blockchain-based access control

development progress.

- OpenZeppelin library offers a set of contracts designed for access control implementation. It utilizes a Role-Based Access Control (RBAC) model, providing developers with capabilities for role definition, assignment, and control. However, it doesn't offer policy management features, thus leaving policy definition and control to the developers [69].
- Dappsys, another practical library, provides a simple implementation of a whitelist, enabling permission granting to addresses for specific contract function calls. Unlike OpenZeppelin, Dappsys offers policy definition and management features, thereby easing the complex process of access control management for developers [70].
- An academic work that aligns with our research is titled "Dynamic Role-Based Access Control Framework for Decentralised Applications (dApps)" [71]. In this work, the authors suggest a framework for managing access control in decentralized apps, emphasizing the dynamic nature of role-based access control. Their work's distinguishing feature is the separation of access control from the business logic and for decentralized governance of access control policies. Although our problem space aligns with theirs, part of the focus of our work is the different architectures we can use for such upgradable systems using specific characteristics of smart contracts.
- An attribute-based access control platform using smart contracts is proposed by the authors of [72]. While they share our perspective of providing access control management as a service, our models differ fundamentally. They propose off-chain execution of policy definition due to the costly nature of XACML creation on Ethereum, resulting in a static model without update capabilities. In contrast, we propose on-chain policy creation with dynamic update capabilities.
- Another academic work closely related to ours is 'DEDACS: Decentralized and dynamic access control for smart contracts in a policy-based manner' [73]. This promotes greater flexibility and efficiency in real-world applications. DEDACS also has limitations, like the inability to dynamically add new attributes after deployment, for which we propose a solution.

All discussed solutions focus on an RBAC, ABAC, or policy-based solution for access control management. In contrast, our proposed solution integrates a hybrid access control model, Role-Centric Access Control, rather than a pure ABAC or RBAC model to offer a more comprehensive solution.

Integrating practical libraries and scholarly research materials facilitates comprehension of current affairs. The proposed method seeks to integrate the advantages of libraries like OpenZeppelin, which is widely used and is known as a best-practice library, and academic ideas to provide a proficient and streamlined approach to access control management within a decentralized environment.

3.3 Blockchain-Based Access Control Management for Off-Chain Platforms

Numerous blockchain-based access control management solutions, such as IoT or healthcare systems, have been designed for off-chain use cases. These solutions often vary significantly depending on the specific platform requirements they are designed for.

While the focus of our discussion is primarily on Ethereum-based access control management for off-chain platforms, it's worth noting the existence of solutions implemented on other blockchains like Hyperledger Fabric in [74] or EOS in [75] which utilizes state channels for improved scalability.

Other solutions deploy a range of access control models beyond RBAC and ABAC, including Organization Based Access Control (OrBAC) [76], access control lists [77], entitlement-based access control [78], capability-based access control [79], [80], Location-Aware Role-Based Access Control [81], and access control matrix [82]. Each model is selected based on the specific requirements and characteristics of the off-chain platform they serve.

One notable solution in this category is FairAccess [76], recognized as one of the pioneering access control systems based on blockchain technology. FairAccess was expressly developed to address the challenges inherent in centralized IoT Access Control Management systems. FairAccess employs an organization-based Access control (OrBAC) framework, conceptualizing IoT settings as distinct entities within the context of a smart city. The authors subsequently introduced FairAccess2.0 [83], a framework that utilizes smart contracts to implement an Attribute-Based Access Control (ABAC) paradigm. The authors of the FairAccess platform implemented the PPDAC Model [84] as an additional measure to improve privacy.

Additional noteworthy off-chain access control management solutions have been proposed in academic literature. For instance, the authors of [85] have proposed an Ethereum-based access control and data sharing model specifically designed for Internet of Things (IoT) devices. Another notable solution is the Attribute-Based Access Control (ABAC) model proposed for smart cities' access control management, which utilizes Ethereum smart contracts [86]. Furthermore, a Blockchain-based Access Control System (BACS) has been proposed in [87], where blockchain node addresses are employed for IoT server authentication and on-chain

policy storage.

Upon conducting a comprehensive review of these proposed solutions, it becomes evident that most of them, particularly those not explicitly developed for smart contracts, exhibit insufficient attention to privacy concerns. In addition, there is a notable absence of comprehensive debates about gas optimization, a critical obstacle in the application of blockchain technology. The proposed method presented in this study aims to mitigate the aforementioned limitations by implementing a Role-Centric Access Control (RCAC) model. This model combines the advantages of Attribute-Based Access Control (ABAC) in terms of flexibility and Role-Based Access Control (RBAC) in terms of simplicity, resulting in a more balanced approach.

CHAPTER 4 AUTOMATED ANALYSIS OF HALO2 CIRCUITS

4.1 Introduction

A *zero-knowledge* proof system is a protocol that allows a *prover* to convince a *verifier* that a certain statement is true without revealing any other information [88]. Recently, a specific type of these proof systems, which are classed as *zk-SNARK*, have gained popularity partly due to their applications in scaling blockchain systems and building privacy-preserving applications.

This popularity comes from three desirable properties of zk-SNARK systems: that they are *zero-knowledge*, that their proofs are *succinct*, and that these protocols are *non-interactive*. A proof system is *zero-knowledge* if the verifier gains no knowledge beyond the correctness of the statement being proved, and it is *succinct* if verifying the proof takes time poly-logarithmic in the computation. The proof system is *non-interactive* if the prover can generate the proof in one round, and any verifier can check this proof and be convinced of the outcome.

A zero-knowledge proof system with all three properties is often called a *zero-knowledge Succinct Non-interactive ARguments of Knowledge* or *zk-SNARK* for short. Example zk-SNARKs include Groth16 [5], Halo [6], and PLONK [2].

The properties of zk-SNARKs described above make zk-SNARKs attractive and useful for blockchain applications. First, zero-knowledge proof systems may be used to enable private and anonymous transactions on these trustless ledgers. For example, Tornado Cash [7] is a zero-knowledge application to anonymize transactions on the Ethereum blockchain [89]. Second, the non-interactive and succinct properties of such a system may be used to scale these systems even when the zero-knowledge property is not desired. That is, succinctness is valuable even when all inputs to the proof system are assumed to be public. Succinct proofs enable scaling of blockchains like Ethereum as computation can be performed off-chain but verified on-chain. These scaling solutions are called *rollups* (also called *validating bridges* [90] or *commit-chains* [91]) and are protocols that conform to this idea of separating execution and verification. Rollups which utilize zero-knowledge proof systems are called *zero-knowledge rollups*¹. Examples of zero-knowledge rollups include the Polygon zkEVM, StarkNet, and Scroll.

¹We note that the term *zero-knowledge rollup* is a misnomer: the *zero-knowledge* part is often not important; more accurately these systems only value the computational integrity proofs generated by the zero-knowledge framework.

zk-SNARKs are often used to prove statements of the format “*For function f and public input x , I know a private witness w such that $f(x, w) = y$* ”. A suitable arithmetization of the function is necessary as the SNARK proves an NP relation, often over a finite field. This can be done directly for simple functions or using a compiler from a Domain-Specific Language (DSL) for more complex functions like those that represent the execution of virtual machines. Examples of zero-knowledge DSLs include Circom [92] and ZoKrates [93]. Using these DSLs enables developers to build zero-knowledge applications by abstracting away the explicit notion of computing the function f described above. There are also zk-SNARK frameworks that do not inherently rely on a DSL for their implementation. One popular example of such libraries is Halo2, introduced by Zcash.

Unsurprisingly, zero-knowledge applications may contain bugs. First, developers writing such zero-knowledge applications in a DSL or using a library like Halo2 may introduce bugs directly. Second, there may be errors in the resulting arithmetization introduced by the DSL compiler (e.g., bugs in Leo² and Noir³). To make matters worse, for complicated applications, these proof systems translate computations to arithmetizations too large to be manually checked.

To overcome this challenge, the zero-knowledge application community has turned to tooling. Automated tools like Ecne⁴, Picus⁵, and QED² [17] exist for checking *Rank-1 Constraint System* (R1CS) arithmetizations. These tools often check various notions of correctness but pay special attention to whether or not the resulting system is *under-constrained*. If it has multiple valid efficiently computed assignments to satisfy the constraints (this is defined formally in Section 4.3). Under-constrained circuits are critical security vulnerabilities that could have allowed user funds to be stolen in Tornado Cash [18] or allowed repeated spending of tokens in the Aztec network [19]. However, both instances have been detected before being exploited and fixed.

Beside the R1CS representation, other representations exist and often build on the PLONK [2] proof system; these are said to use *PLONKish* arithmetization. Oversimplifying, this representation transforms a computation into a circuit consisting of a set of gates. Each circuit is represented as a table of values and polynomials constraining the table’s rows. A more detailed description is provided in Section 6.3. To our knowledge, no automated tools exist to check applications built using PLONKish arithmetizations.

This research proposes a tool for analyzing one of the most popular PLONKish arithmetiza-

²<https://github.com/AleoHQ/leo/issues/2042>

³<https://github.com/noir-lang/noir/issues/358>

⁴<https://github.com/franklynwang/EcneProject>

⁵<https://github.com/Veridise/Picus>

tions, Halo2 circuits, using the CVC5 SMT solver for detecting under and over constrained circuits.

4.2 Halo2

Halo2 is a popular zero-knowledge proof system implemented in Rust. It uses PLONKish arithmetization to express functions or applications as circuits, as originally introduced by PLONK [2]. PLONK has various extensions (e.g., HyperPLONK [94], TurboPLONK [95], and UltraPLONK [96]) which adds more expressive custom gates to the proof system, combines PLONK with so-called *lookup* tables [97], and provides more efficient proving mechanisms for the same type of arithmetization.

PLONKish circuits are represented as tables of values from a finite field. Entries of the table are called *cells*. Circuits are made up of gates, which are constraints applied to sub-tables (a set of cells) across the table. Table columns have three main types: *instance*, *advice*, and *fixed*. *Instance* columns store public input, *advice* columns store private witness (private inputs and intermediate values used by the prover in the circuit), and *fixed* columns store fixed values that are part of the description of the circuit. The values of the table are constrained by three types of constraints: *gate constraints* and *equality constraints*, which we now define, and those from *lookups*, which we describe in Section 4.4. Gate constraints are generic polynomial constraints defined by the circuit designer. We will use $g(\langle w \rangle)$ to denote a gate constraint on $\langle w \rangle$, which represents an encoding of the witness w along with (unnamed) public inputs. Each gate constraint applies to a number of adjacent rows of the gate. Therefore a circuit whose table representation has several rows may have several polynomial constraints, all of which should be simultaneously satisfied. Equality constraints (a.k.a. *permutation constraints* or *copy constraints*) can enforce specific group of cells to have equal values across rows and columns. We refer to the table and its corresponding constraints as the *constraint system*.

With Halo2, gates are enabled by *selector* variables (or simply *selectors*). A *selector* is a binary variable that is multiplied to the polynomial constraint for each row: it is 0 if the gate is off or 1 if it is on. We will use s to denote a selector variable, and as a result, a polynomial constraint g on a gate is more accurately⁶ represented as $f(\langle w \rangle, s) = s \cdot g(\langle w \rangle)$ for a witness w .

We say that a polynomial *vanishes* if it evaluates to 0 over all rows. All polynomial constraints in a Halo2 proof system should vanish over all rows for a valid witness and public input pair. This is done either by setting the irrelevant selector variables to 0, or by providing (possibly

⁶Recall that a gate is technically applied to all rows, and so this is actually the Hadamard product.

secret) assignments to the table cells that result in the polynomial's evaluation to 0.

We illustrate the computation of a Halo2 gate in the following example. Suppose we have the following polynomial gate constraint, whose table is shown in Table 4.1:

$$f(\langle a, b, C \rangle, s) = s \cdot (C \cdot a - b). \quad (4.1)$$

The variables have the following interpretations: s is the selector (if it is 0 then the polynomial evaluates to 0), a is the input of the gate, b is the output of the gate, and C is a parameter used to configure the gate (a constant in the circuit). The gate computes the following relation: $b = C \cdot a$ (so that the polynomial is 0 even when $s = 1$). Thus the gate in this example is a multiplication by a constant gate.

A *region* in Halo2 is a logical collection of cells with a set of enabled gates (as indicated by the selectors). This work operates by collecting all the gates and regions using a custom implementation of the Halo2 **Layouter** trait, which normally is responsible for the place-and-route behaviour of compiling the circuit. This functionality is responsible for mapping the regions to rows to minimize the number of required rows to enforce the constraints; it essentially solves a packing problem. This is done because there is a limit to the number of rows used, though we can increase the number of columns at will by adding some complexity to the proof system. Our implementation of this trait, **AnalyticalLayouter**, will simply record the details of every region. Since we never need to actually generate a proof, an optimal packing is not necessary.

Finally, we note that the Halo2 Rust library offers a **Circuit** trait, the standard interface that any circuit object should implement. We are also interested in one method of the **Circuit** trait, **synthesize**. The **synthesize** function is responsible both for creating the constraint system and witness generation. This is relevant for Section 4.3.

Selector (s)	Instance (b)	Advice (a)	Fixed (C)
1	15	5	3
<i>Established at circuit configuration.</i>	<i>Public input; known to prover and verifier.</i>	<i>Private witness; known only to prover</i>	<i>Public constant; known to prover. and verifier.</i>

Table 4.1 A very small Halo2 circuit's table, filled with real values. The corresponding polynomial constraint is equation (4.1). The table is filled to demonstrate that the prover knows an assignment $a = 5$ such that $C \cdot a = 3 \cdot 5 = 15 = b$ (which implies $C \cdot a - b = 0$, as required). That is, the prover knows the necessary factor a such that $C \cdot a = b$. Since $s = 1$, the gate is on.

4.3 Under- and Over-constrained Circuits

In the context of PLONKish circuits, we define the properties of being under- and over-constrained, before showing how to detect such circuits using an SMT solver.

Let C be a PLONKish circuit. We call C *under-constrained* if there exist an assignment x to instance columns of C , and two set of assignments w and w' for its advice columns, where both $\{x, w\}$ and $\{x, w'\}$ satisfy constraints of C .

Being under-constrained is a usually a sign of error by circuit designers; most likely, the arithmetization is not faithful to the intended function. One common example is calculating value of a cell in the table as part of witness generation in `synthesizer` but failing to select (or activate) the right gates to enforce a constraint over this cell. This may enable a malicious prover to create valid proofs that will be accepted by the verifier but should not be. Recall that under-constrained circuits have been a source of errors for real-world protocols. While some circuits may be under-constrained intentionally, we assume that users of this tool expect their circuit to have a single assignment for every valid input.

A PLONKish circuit C is *over-constrained* if for some assignment x to instance columns of C , no assignments to the advice columns of C enable the system to have a solution, but the developer expects there to be one. Note that the last part of the definition is necessary, as otherwise some circuits would always be over-constrained. As a trivial example, consider a circuit that states that for any positive integer x as input, there are two (distinct) advice columns entries that are positive integer and add up to x . For $x \geq 2$, this circuit can be satisfied (in multiple ways), but for $x = 1$ this is not the case, and it would not be meaningful to call the circuit over-constrained for this input value.

Example 4.3.1. We provide an example where the aim is to check if a public input x is in the set $\{0, 1, 2, 3\}$. The code aims to implement the following constraints over a suitable Halo2 table:

$$b_0 \cdot (b_0 - 1) = 0 \tag{4.2}$$

$$b_1 \cdot (b_1 - 1) = 0 \tag{4.3}$$

$$b_0 + 2 \cdot b_1 - x = 0 \tag{4.4}$$

This set of constraints is uniquely satisfied if $x \in \{0, 1, 2, 3\}$, but is not satisfiable otherwise.

We provide an example of an under-constrained implementation of the previous system in Figure 4.1. The `meta` object is an instance of a `ConstraintSystem`, which creates Halo2 gates. In this example, lines 1-5 create the gate implementing constraint (4.2), lines 6-10

create the gate implementing constraint (4.3), and lines 11-17 create the gate implementing constraint (4.4). Line 2 gets a variable `b1` to be used in the gate from the advice column in the current row⁷ because of the function `cur` called on the `Rotation` object. Line 3 gets the selector value from the configuration, that is, if this gate will be used. Finally line 4 encodes the expression $d \cdot a \cdot (1 - a)$, making new copies of `a` as required by Rust. The expression on line 4 will be assumed to be equal to zero in the Halo2 library.

Lines 1-5 and 6-10 are similar; so similar, that the developer could copy-and-paste lines 1-5 to get lines 6-10 with small changes. However, in this example, the necessary modifications are not completed: the gate's name was updated on line 6, but the variable name `b1` was not updated on line 7. As a result, lines 6-10 actually encode the same constraint as lines 1-5.

Line 16 encodes the expression $d \cdot (a + 2b - c)$ which represents $b_0 + 2b_1 = x$ as the proof system will enforce that the expression is equal to zero, along with the selector factor `d`. Note that although line 14 uses `x` from an advice column, it can be enforced to have the same value as the public input elsewhere in the code, e.g., by using the `constrain_instance` function and the appropriate public instance column, elsewhere in the code.

Throughout this example, we can ignore the selector values `d` as we will assume all gates are used and this is the only configuration of interest. With that in mind, we can see that the system will be under-constrained: when we set $x = 3$, there are two ways to assign values to the advice columns so that the system is satisfied: $\{b_0 = 1, b_1 = 1\}$ and $\{b_0 = 3, b_1 = 0\}$. In short, the hasty developer did not constraint b_0 to be in $\{0, 1\}$.

```

1 meta.create_gate("b1_binary_check", |meta| {
2     let a = meta.query_advice(b1, Rotation::cur());
3     let d = meta.query_selector(s);
4     vec![d * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
5 });
6 meta.create_gate("b0_binary_check", |meta| {
7     let a = meta.query_advice(b1, Rotation::cur());
8     let d = meta.query_selector(s);
9     vec![d * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
10 });
11 meta.create_gate("equality", |meta| {
12     let a = meta.query_advice(b0, Rotation::cur());
13     let b = meta.query_advice(b1, Rotation::cur());
14     let c = meta.query_advice(x, Rotation::cur());

```

⁷If the rotation referenced `prev` or `next` or something else, we would change the index of the corresponding variable in the generated output to reference the concrete cell, rather than the relative one. For example, if B-7 is the variable for the 7th row of the B column but the rotation was `next`, we add a constraint that B-7=B-8.

```

15     let d = meta.query_selector(s);
16     vec![d * (a + Expression::Constant(Fr::from(2)) * b - c)]
17 });

```

Listing 4.1 An erroneously under-constrained set of gates in Halo2. The variable **b1** on line 7 should in fact be **b0**.

Soundness and Integrity Issues in Under-constrained Circuits

The soundness of a zero-knowledge proof system is critical, ensuring that it is computationally infeasible for a dishonest prover to successfully convince a verifier of a false statement. Under-constrained circuits compromise this property by allowing multiple valid assignments for a given input, which can be exploited by dishonest provers.

Impact on Circuit Integrity Under-constrained circuits allow malicious provers to create valid proofs for conditions that are actually invalid, thereby jeopardizing the verifier’s trust. This is not only a breach of soundness but also of integrity, as the proof system fails to enforce the truthful representation of knowledge. For example, if a circuit intended to verify prime numbers is under-constrained, a prover could manipulate the circuit to accept a composite number as prime, producing a valid proof for an incorrect claim.

To prevent such scenarios, it is essential for circuit designers to meticulously define and enforce constraints that uniquely tie inputs to their correct outputs, ensuring that every computational step within the circuit is accounted for and appropriately constrained. This rigorous approach to circuit design and verification prevents ambiguity and restricts the space of valid proofs to those that genuinely correspond to truthful statements.

We now describe how to identify an under-constrained or over-constrained system in Halo2 using `cvc5` [20]. We use the `cvc5` solver as there is a finite field solver incorporated into it [98], facilitating a more direct conversion of polynomials to solver constraints. We focus on detecting under-constrained systems, but are able to check over-constrained systems for free. The idea is to extract the polynomial constraints, convert them to a model that can be checked for satisfiability in an SMT solver, and then determine if there are multiple satisfying models.

4.4 From Halo2 to SMT

We outline the conversion of Halo2 constraints into the format required by an SMT solver.

First, we build constraints based on the polynomials built in Halo2. Within our configuration, we can find all gates $\mathcal{G}_i$ within a region. Let $\mathcal{G}$ be the set of all gates $\mathcal{G}_i$ (that is, $\mathcal{G} = \{\mathcal{G}_i \mid \mathcal{G}_i \text{ is a gate}\}$). For each gate $\mathcal{G}_i$, we can collect each polynomial $\mathcal{P}_{i,j}$ defined within it: let $\mathcal{P}_i = \{\mathcal{P}_{i,j} \mid \mathcal{P}_{i,j} \text{ is a polynomial in } \mathcal{G}_i\}$; finally, let $\mathcal{P} = \cup_i \mathcal{P}_i$.

The polynomials are created in two major steps within the proof system: circuit configuration and when circuits are synthesized. The result provides a set of polynomials based on selectors set during these steps, as well as constraints that enforce values and the relationship between values in the table. Specifically, we enforce values given from a particular input and any witnesses.

Next, we build SMT constraints based on the polynomials in the set $\mathcal{P}$. For all polynomials $\mathcal{P}_{i,j} \in \mathcal{P}$ collected from constraint system, we add an **assert** statement, where $\mathcal{P}_{i,j}$ is converted to an expression accepted by the SMT input language (SMT-LIB [99]). An example of an equality constraint is shown in Figure 4.2. A variable **A-i-j-k** is the value of column A in region i , row j , and column k ; **A-1-1-1** is the first entry of the first row and column in the first region.

```

1 (set-logic QF_FF)
2 (declare-fun A-1-1-1 () (_ FiniteField 307))
3 (assert (= A-1-1-1 (as ff0 (_ FiniteField 307))))

```

Listing 4.2 Example SMT output generated by our tool in the format of SMT-LIB [99]. The finite field logic (Quantifier Free Finite Field) is declared on line 1. In this example, the field is over the prime 307. Line 2 declares a variable for the model, and line 3 declares a sample equality constraint that **A-1-1-1** = 0, as **ff0** represents zero in the finite field (not shown).

We create an instance **S** of an SMT solver. To this solver, we add each of the assertions created from the set $\mathcal{P}_{i,j}$, and thus the initial set of constraints in the solver is

$$\mathcal{C} = \bigwedge_{i,j} (\mathcal{P}_{i,j} = 0),$$

along with all copy constraints, constraints necessary to enforce equality due to rotations, and constraints related to lookups.

Halo2 handles lookup tables by adding a fixed column for each column in the lookup table. Within the constraint system, there is a **lookups** component which has an **argument** field, which comprises two vectors: **input_expressions** and **table_expressions**. The **input_expressions** vector determines which advice columns are constrained to occur in the lookup table. The **table_expressions** vector includes the column indexes of the fixed columns, which should equal the advice columns extracted from **input_expressions**. Us-

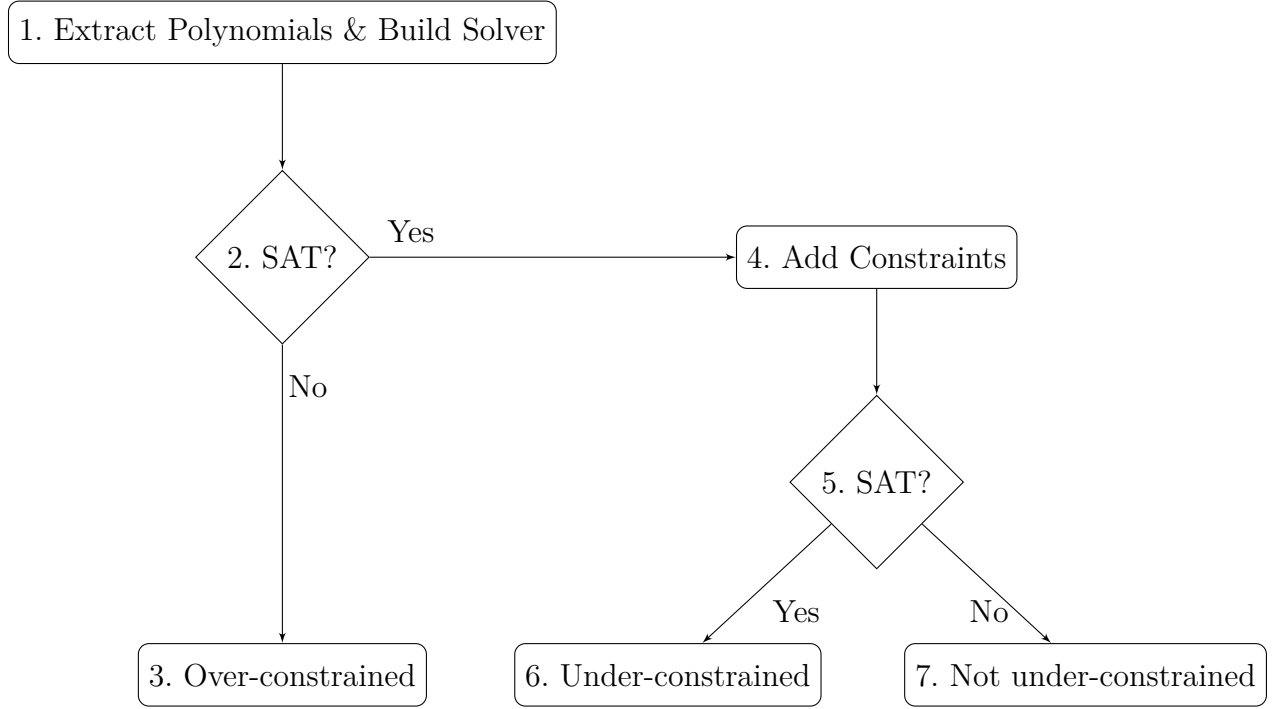


Figure 4.1 The process used to detect under- and over-constrained systems in our tool.

ing these vectors, we can determine which advice entry should occur in which fixed column index. Then, we implement the search in the lookup table as a constraint. For that, we need to have the actual values for the lookup table columns. We can extract this by running the Halo2 mock prover, which gives us a two-dimensional vector (**Fixes**) that includes each fixed column's values. The lookup can then be modelled as a constraint as a disjunction of conjunctions. Each conjunction is the constraint required by the fixed columns, mapped to entries in the advice columns. Since any values satisfying such a constraint is valid, we take the disjunction over all possible valid conjunctions. We illustrate via examples. A simple range look up is shown in Table 4.2; in this case, the inner conjunction does not exist; it has only one term. A more complicated example is shown in Table 4.3, which illustrates the need for the internal conjunction.

We now describe the process used to determine if a system is under- or over-constrained. This process is illustrated in Figure 4.1. The solver S is queried to see if it can find a satisfying assignment, and if so, a model that satisfies S (i.e., an assignment to the variables of the polynomials) is generated. If no such model is found, the system is *over-constrained* (and hence not under-constrained). If a model has been obtained, the system may not be under-constrained. We seek to find another, different model to also satisfy the system. Therefore, we add a set of constraints that ensure that at least one variable in the model must be

Advice A_0	Fixed F_0
$\vdots$	c89efdaa
$\vdots$	7f8b6b08
a_i	2a80e1ef
$\vdots$	13600b29
$\vdots$	ceebf77a

Table 4.2 A small range check modelled as a lookup in a Halo2 circuit table (only the relevant columns are shown). In this case, we want the i row of the advice column A_0 , a_i to be one of the values in the only fixed column F_0 . We add the following constraint to the solver: $(a_i = c89efdaa) \vee (a_i = 7f8b6b08) \vee \dots \vee (a_i = ceebf77a)$.

Advice A_0	Advice A_1	Advice A_2	Fixed F_0	Fixed F_1	Fixed F_2
$\vdots$	$\vdots$	$\vdots$	000	000	000
$\vdots$	$\vdots$	$\vdots$	000	001	001
$a_{0,i}$	$a_{1,i}$	$a_{2,i}$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	111	111	000

Table 4.3 A lookup in a Halo2 circuit table with constraints on the fixed columns (only the relevant columns are shown). In this case, the entry of F_2 at row i is the bit-wise exclusive or function (XOR) applied to the values of row i in F_0 and F_1 . In this case, we want the i row of the advice columns A_0 , A_1 , and A_2 to be a valid XOR for entries which have three bits. We add the following constraint to the solver: $(a_{0,i} = 000 \wedge a_{1,i} = 000 \wedge a_{2,i} = 000) \vee (a_{0,i} = 000 \wedge a_{1,i} = 001 \wedge a_{2,i} = 001) \vee \dots \vee (a_{0,i} = 111 \wedge a_{1,i} = 111 \wedge a_{2,i} = 000)$.

different.

Let $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ be variables that are constraining public inputs, and let $\{\beta_1, \beta_2, \dots, \beta_m\}$ be the variables that do not constrain public inputs. Let $\{\hat{\alpha}_1, \hat{\alpha}_2, \dots, \hat{\alpha}_n\}$ (respectively $\{\hat{\beta}_1, \hat{\beta}_2, \dots, \hat{\beta}_m\}$) be the set of values assigned by the satisfying model to the constraining (respectively non-constraining) variables. Then we have that $\alpha_i = \hat{\alpha}_i$ for $1 \leq i \leq n$ and $\beta_j = \hat{\beta}_j$ for $1 \leq j \leq m$ in the model. We create a new solver $\mathbf{S}'$ whose constraint set is

$$\mathcal{C}' = \{\mathcal{C} \wedge (\beta_1 \neq \hat{\beta}_1 \vee \beta_2 \neq \hat{\beta}_2 \vee \dots \vee \beta_m \neq \hat{\beta}_m)\} \quad (4.5)$$

If $\mathbf{S}'$ is satisfiable, we have proved that the circuit is under-constrained.

4.5 Analyzer Design and Implementation

In this section, we explain the design and implementation details of the analyzer. As explained, we must extract all the constraints defined on a circuit's table. Different functionalities imply these constraints in the circuit.

Figure 4.2 represents the analyzer's design and its relationship with halo2. The analyzer is initialized from a circuit and has an instance of the circuit's `ConstraintSystem`, including the custom gates. It also has an instance of `AnalyticalLayouter`, a custom implementation of halo2 `Layouter`. It enables us to call the `synthesize` function, a standard function of any circuit that implements halo2's `Circuit` trait. An instance of the `AnalyticalLayouter` is passed to the `synthesize` function to extract and maintain any information we need to analyze the circuit. The `AnalyticalLayouter` itself has a vector of `AnalyticalShapes`, a custom implementation of halo2's `RegionLayouter`. Together with the `ConstraintSystem`, these components allow us to model the whole circuit as a set of constraints to be analyzed in the SMT solver. We also need to have the exact stored values in the fixed columns in the circuit's table, so we use `MockProver` to instantiate the table and fill the fixed columns, and vectors of fixed columns are passed to the analyzer.

We can extract the gates from the `ConstraintSystem`. To analyze if a circuit is under-constrained using an SMT solver, we will need to extract, in addition to the gates and regions, all relations between cells, including copy constraints, constraints related to lookups, and the constraints related to exposing some specific cells to public inputs.

To achieve these, we have added some custom attributes to the analytical components: In `AnalyticalLayouter`:

```
1 pub struct AnalyticLayouter<F: Field> {
```

```

2   pub regions: Vec<AnalyticalShape>,
3   _ph: PhantomData<F>,
4   pub eq_table: HashMap<String, String>,
5 }

```

Listing 4.3 AnalyticLayouter Struct

eq_table in AnalyticLayouter keeps the pairs of advice columns that should be equal as part of constraining an advice column to another advice column which is constrained to be equal to a public input. This list is updated by each call to constrain_instance() function. Figure 6.1 shows the structure of AnalyticalShape.

```

1 pub struct AnalyticalShape {
2     pub name: String,
3     pub region_index: RegionIndex,
4     pub columns: HashSet<(RegionColumn, Rotation)>,
5     pub row_count: usize,
6
7     pub enabled_selectors: HashSet<String>,
8
9     pub advice_eq_table: HashMap<String, String>,
10    pub eq_table: HashMap<String, String>,
11 }

```

Listing 4.4 AnalyticalShape Struct

A list of enabled_selectors on each region is tracked and updated on each call to enable_selector() function. advice_eq_table in AnalyticalShape keeps the pairs of advice columns that should be equal as part of permutation of advice columns. This list is updated by each call to constrain_equal() function.

eq_table in AnalyticalShape keeps the pairs of advice columns and instance columns that should be equal. This list is updated by each call to assign_advice_from_instance() function.

The analyze_underconstrained() function extracts all constraints in several steps:

decompose_polynomial() This function first converts all the polynomials from custom gates defined in ConstraintSystem enabled by selectors in any region into SMT constraints by calling the decompose_expression() function. Each region (of type AnalyticalShape), along with other attributes, has a row_count, and a set of enabled_selectors. The decompose_expression function takes as input a region index, a polynomial, a row number, and a set of enabled selectors in the specified region. This function is called on each row in each region in AnalyticLayouter. In the next step, decompose_polynomial extracts constraints related to

lookups. `halo2` handles the look-up tables by adding a fixed column for each column in the look-up table. In the constraint System, there is a specific lookups part which has an argument field; the argument includes two vectors: `input_expressions`, which determines which advice columns are constrained to occur in the look-up table. The second vector is the `table_expressions`, which includes the column indexes of the fixed columns, which should accordingly equal the advice columns extracted from `input_expressions`. So, we have extracted which advice index should occur in which fixed column index. `decompose_polynomial` goes over `input_expressions` and calls `decompose_expression` to extract which advice columns are constrained to occur in the look-up table, and then goes over the `table_expressions`, which includes the column indexes of the fixed columns, which should accordingly equal the advice columns extracted from `input_expressions`. The next thing is to implement the search in the look-up table. For that, we needed to have the actual values that the look-up table (now a number of fixed columns) is filled with. We could extract this by running the mock prover, which gives us a two-dimensional vector (`Fixes`) that includes each fixed column's values. At this point, we have extracted all the constraints from custom gates and lookups.

`analyze_underconstrained` continues by adding the remaining constraints that are not directly implied by custom gates:

- the constraints that imply two advice columns (at specific offsets and regions) have equal values,
- the constraints that imply two an advice column and an instance column (at specific offsets and regions) have equal values
- and the constraints that imply that some instance columns (at specific offsets and regions) are equal to some public inputs.

All the extracted constraints are written into the SMT2 file in the above steps. These operations are handled by other components designed for SMT solver management.

After extracting all constraints, the `uniqueness_assertion()` is responsible for repeatedly finding a model that satisfies the constraints, and such a model exists, trying to find another model by fixing instance columns and changing at least one of the advice columns. If such a model exists, we can say that the circuit is under-constrained. Otherwise, we add a set of constraints to ensure this model is not generated again and look for another model that satisfies the constraints. The solver stops if there is no other model that satisfies the constraints, meaning that the circuit is not under-constrained.

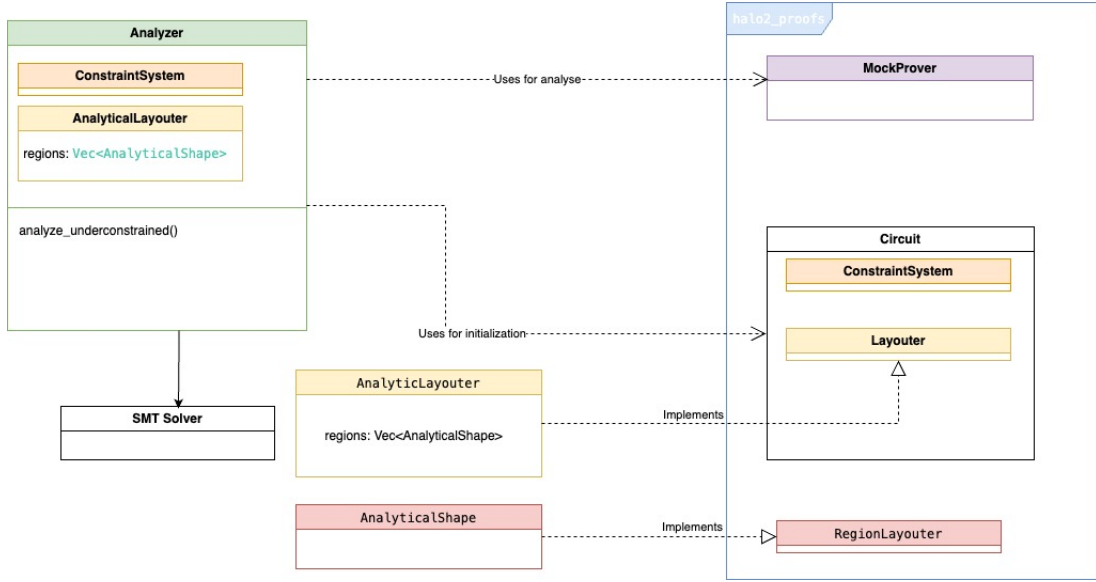


Figure 4.2 Analyzer Design

We present key pseudocode algorithms used in the analysis of circuit constraints and polynomial expressions as explained before. These algorithms help in converting circuit logic into a form suitable for SMT solvers, ensuring correctness and consistency in cryptographic circuits.

Algorithm 1 describes the process of converting polynomials from circuit gates into SMT-LIB assertions. It ensures each polynomial evaluates to zero and handles lookup tables by comparing inputs with table values.

Algorithm 2 focuses on handling copy constraints, which require equality between variables in different regions of the circuit. It generates SMT assertions to ensure these constraints are maintained.

Algorithm 3 deals with decomposing various types of expressions (constants, selectors, advice columns, and recursive expressions) into SMT-LIB terms. It handles both simple fixed columns and complex recursive operations like sums, products, and scaling.

These algorithms form the backbone of the constraint analysis and ensure the circuit is prepared correctly for verification by an SMT solver.

4.6 Fuzzing

Our system can be configured to also generate multiple public inputs. We ask for a satisfying assignment to the circuit, and if one is found, we ask if another satisfying assignment is

Algorithm 1 DecomposePolynomial

```

1: Input: Circuit, Printer, FixedColumns
2: for Each Region in Circuit do
3:   for Each Row in Region do
4:     for Each Gate in Circuit do
5:       for Each Polynomial in the Gate do
6:         Decompose Polynomial into SMT-LIB format
7:         Write SMT-LIB assertion that Polynomial equals 0
8:       end for
9:     end for
10:    for Each Lookup in Circuit do
11:      Collect input expressions from Circuit
12:      Decompose each expression into SMT-LIB format
13:      for Each Row in Lookup Table do
14:        Compare input expressions with corresponding table values
15:        Generate equality assertions for each comparison
16:      end for
17:      Write SMT-LIB assertion that at least one row satisfies lookup
18:    end for
19:  end for
20: end for

```

possible for this particular public input assignment. If it is, we stop, as we have shown that the circuit is under-constrained. If not, we check if the circuit can be satisfied by another public input, and repeat until no more assignments can be found or we run out of time. This way the system acts as a sort of “fuzzer,” allowing developers to explore assignments. We outline this process now.

Let $\mathcal{C}$ be the constraints for the circuit, as defined in Section 4.3. Then, we iterate over possible assignments to $\mathcal{C}$ to try to find a public input for which the circuit would be under-constrained.

Algorithm 2 Handling Copy Constraints

```

1: Input: Circuit regions with advice and equality tables
2: Output: Assertions for SMT Solver
3: for Each Region in the Circuit Layout do
4:   for Each pair of variables in the region do
5:     Record both variables
6:     Write SMT-LIB assertion that variables are equals
7:   end for
8: end for
9: return SMT Assertions for all regions

```

Algorithm 3 DecomposeExpression

```

1: Input: Expression, RegionID, RowID
2: Output: SMT-LIB Term
3: // Fixed, Selector, and Constant Expressions
4: if Expression is Constant or Selector or Fixed Column then
5:   Convert Expression to SMT-LIB constant type
6:   return SMT-LIB term
7: end if
8: // Advice and Instance Columns
9: if Expression is Advice or Instance Column then
10:   Generate SMT-LIB term for Variable using RegionID and RowID
11:   Write SMT-LIB variable declaration
12:   return SMT-LIB term
13: end if
14: // Recursive Operations
15: if Expression is Sum, Product, Negated, or Scaled then
16:   Recursively decompose inner Expressions
17:   Add result to the SMT-LIB term
18: end if
19: return SMT-LIB term

```

Let $\mathcal{C}_0 = \mathcal{C}$. Then, at iteration $i \geq 0$, we ask a solver $\mathbf{S}$ to determine if $\mathcal{C}_i$ is satisfiable; if not, the circuit is over-constrained or we have exhausted all valid public inputs. The loop stops if $\mathcal{C}_i$ is not satisfiable.

We may therefore assume $\mathcal{C}_i$ is satisfiable with a public input assignment set $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, \dots, \hat{\alpha}_n^i\}$ and witness assignment set $\{\hat{\beta}_1^i, \hat{\beta}_2^i, \dots, \hat{\beta}_m^i\}$. Then we ask a solver to determine if

$$\mathcal{C}'_i = \{\mathcal{C}_i \wedge (\beta_1 \neq \hat{\beta}_1^i \vee \beta_2 \neq \hat{\beta}_2^i \vee \dots \vee \beta_m \neq \hat{\beta}_m^i) \wedge (\alpha_1 = \hat{\alpha}_1^i \wedge \alpha_2 = \hat{\alpha}_2^i \wedge \dots \wedge \alpha_n = \hat{\alpha}_n^i)\} \quad (4.6)$$

is satisfiable. If $\mathcal{C}'_i$ is also satisfiable, we have shown that the circuit is under-constrained as we have two assignments to private witness for a single assignment to the variables of $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, \dots, \hat{\alpha}_n^i\}$.

Otherwise, we determine a new public input to try for constraint set $\mathcal{C}_{i+1}$ by preventing any future solver calls from using the public input assignment $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, \dots, \hat{\alpha}_n^i\}$. Explicitly, we have

$$\mathcal{C}_{i+1} = \{\mathcal{C}_i \wedge (\alpha_1 \neq \hat{\alpha}_1^i \vee \alpha_2 \neq \hat{\alpha}_2^i \vee \dots \vee \alpha_n \neq \hat{\alpha}_n^i)\}. \quad (4.7)$$

Now the process repeats with $\mathcal{C}_{i+1}$.

4.7 General Property Checking

The use of an SMT solver to detect under- and over-constrained circuits allows Rust-savvy users to check other properties in some situations.

As an example, we illustrate another way we could find that the previous example is under-constrained. Suppose that instead of asking for two satisfying models, we are conscious of the need for b_0 and b_1 to be binary variables and want to see if it is ever the case that a satisfying model would violate these constraints. In this case, we could ask the SMT solver to take the constraint set $\mathcal{C}$ and also add a constraint $\mathcal{B}$ which says $b_0 \neq 1 \wedge b_0 \neq 0 \wedge b_1 \neq 0 \wedge b_1 \neq 1$. Assuming the solver does not time out, the result is either *satisfiable* or *unsatisfiable*. If $\mathcal{C} \wedge \mathcal{B}$ is satisfiable, it is not the case that both b_0 and b_1 are always binary variables. If the solver finds that these sets of constraints are unsatisfiable, this means that it is always the case that b_0 and b_1 are binary variables (assuming $\mathcal{C}$ is satisfiable on its own). If the SMT solver is unable to find a satisfiable example, the answer may not be known.

4.8 Discussion

We now discuss the limitations and future directions for this work.

First, the use of an SMT solver likely means that this approach will not scale. Future work is required to generate example circuits or find real-world circuits that can be analyzed, in order to determine an approximate number of constraints and variables that can be checked by the solver before timing out. This may still be useful, as our approach can be applied to sub-circuits or gadgets of an arithmetized function. Repositories of circuits like Circom⁸ (used to create the ZKBENCH circuits of [17]) or those used to evaluate Ecne and Picus, do not exist for Halo2 to provide a corpus of circuits to evaluate our tool on. In fact, once a corpus of Halo2 circuits is collected, one could compare the approach of [17] to our approach. Another option is to apply this tool to the open-source collection of Halo2 circuits in the Privacy-Scaling Exploration zkEVM⁹.

⁸<https://github.com/iden3/circomlib>

⁹<https://github.com/privacy-scaling-explorations/zkevm-circuits>

k -bits	time (ms)	k -bits	time (ms)	k -bits	time (ms)
2	18.940541	16	71.075625	64	730.086875
4	28.063583	32	161.558958	128	4478.517416
8	36.888				

Table 4.4 Run times for the analysis in Section 4.3 on generalized circuits of example 4.3.1.

Pilspector¹⁰ uses SMT solvers to analyze state machines written in the Polynomial Identity Language (PIL) language. It looks for non-determinism in PIL description, which is used by the eSTARK proof system [100]. Another interesting direction to consider is to compare their approach to encoding lookups with the approach taken by this tool.

As a starting point for the evaluation, we ran our tool on generalizations of Example 4.3.1. We analyzed circuits for decomposing numbers into k bits for $k \in \{2, 4, 8, 16, 32, 64, 128\}$. Our evaluation chooses a random assignment to the public input and looks for two witness assignments (using the fuzzing approach but limited to a single iteration). The run times are reported in Table 4.4; they were computed using an Apple MacBook Pro with an M1 Pro processor with ten cores and 16GB of RAM. The time increases with the number of bits in the decomposition, which corresponds (roughly) to the number of variables in the model.

Second, there are likely additional issues for Halo2 that can be detected automatically. A collection of bugs related to zero-knowledge bugs is being maintained¹¹, though not all may apply to Halo2 application developers or the Halo2 proof system itself. We could also check that best practices are used in Halo2 programming, adding functionality akin to a linter in future work.

Another limitation of the current analyzer is that it is implemented for a specific version of PSE halo2. Future work can be a generic implementation of the analyzer, which has no dependency on any version of halo2 so to analyze broader versions of halo2 that follow some minimum standard data structure.

¹⁰<https://github.com/Schaeff/pilspector/>

¹¹<https://github.com/0xPARC/zk-bug-tracker>

CHAPTER 5 HALO2 ANALYZER: STATIC DESIGN AND CYCLE NOTION-BASED OPTIMIZATION

5.1 Introduction

The development of cryptographic circuits underpins the security of modern cryptographic systems. Our previous work, as outlined in Chapter 4, introduced an analyzer tool for halo2 circuits, leveraging a custom implementation of halo2 traits along with some data from running a mock prover on a circuit to extract circuit constraints. Building upon this foundation, this chapter introduces a significant enhancement to the halo2 analyzer, characterized by a novel static design and an accompanying optimization technique for the SMT solver rooted in the cycle notion concept.

The motivation for this advancement stems from the intrinsic limitations identified in the dynamic approach of the initial analyzer version, due to essential dependency on running the mock prover. Recognizing the potential for efficiency gains and the need for a more robust analysis framework, we embarked on developing a static analysis method that removes the dependency on runtime elements, thereby facilitating a more direct and streamlined constraint extraction process.

The contributions of this work are twofold: the transition to a static analysis model and the development of a cycle notion-based optimization technique, which together have achieved an average performance improvement of approximately 45.44% in SMT solver efficiency based on execution time. This significant improvement is a testament to the proposed methodologies' efficacy.

The structure of this chapter is methodically designed to guide the reader through the evolution of the halo2 analyzer. Section 5.2 revisits the foundational elements that underpin the initial design and sets the stage for the new developments. Section 5.3 details the architecture and rationale behind the static design of the analyzer. In Section 5.4, the cycle notion-based optimization technique is presented, showcasing the innovative strategies employed to refine solver performance. Section 5.5 provides a comparative evaluation between the two versions of the analyzer, underscoring the enhancements achieved through the new design. The chapter concludes in Section 6.5 with a synthesis of findings and a forward-looking perspective on future research directions.

5.2 Halo2 Circuits and the Analyzer

In this section, we provide a background on the traits and struct in halo2 and the previous version of the halo2 analyzer, like the Assignment trait, the Assembly struct, and the cycle notion. halo2 has several traits that are used to implement the circuits. There are also two important functions in the development of a halo2 circuit: `configure` and `synthesize`. The `configure` function is responsible for setting up the structure of the circuit. It establishes the gates, which are a fundamental building block of a circuit. In addition to this, `configure` also defines the constraints for the circuit. Constraints can be thought of as the rules or conditions that the circuit must adhere to. They can dictate how data flows through the circuit, or limit the possible states the circuit can be in. On the other hand, the `synthesize` function is responsible for assigning data to these constraints. Specifically, it assigns witness and public data. In the context of halo2 and similar cryptographic protocols, witness data typically refers to private inputs and intermediate values that are known to the prover but not to the verifier. Public data, as the name suggests, is known to both the prover and the verifier.

In summary, `configure` sets up the structure and rules of the circuit, and `synthesize` assigns the data that will be processed by the circuit. Both functions are crucial in the development of a halo2 circuit. The Layouter trait is extensively used for extracting the constraints in the previous version of the halo2 analyzer. The design used a custom implementation of Layouter and RegionLayouter to be passed as a parameter to the `synthesize` function. The constraints are extracted as the `synthesize` function executes. The constraints are then used to generate the copy constraints for the SMT solver.

The new design, as shown in Figure 5.1, is based on the static Assignment of the halo2 circuits.

5.3 Analyzer Design

The redesigned halo2 analyzer represents an important improvement in the analysis of halo2 circuits. Transitioning from a dynamic to a static analysis approach, this new version eliminates the dependency on executing a mock prover and processing public inputs, a necessity in the previous model. This advancement enables the analyzer to statically interpret circuit configurations, leading to a more efficient and direct extraction of constraints.

At the heart of this design is the Analyzable structure, a custom implementation of one of the halo2 structures, the Assignment struct, and its trait that encapsulates the circuit's configuration and synthesis process. The implementation of the Assignment trait within the

Analyzable structure is pivotal, enabling the static assembly of circuits. This approach not only streamlines the analysis process but also enhances the solver's efficiency by providing direct access to a comprehensive set of constraints without the overhead of runtime execution.

A key component of the Analyzable structure is its ability to interpret and organize the circuit layout statically, leveraging the Assignment trait to map each component and constraint within the circuit effectively. This process involves a detailed analysis of the circuit's gates, wires, and constraints, translating them into a format readily understandable by the SMT solver.

The introduction of a cycle notion-based optimization technique further distinguishes the new analyzer. By identifying and exploiting the cyclical relationships inherent in the circuit's assembly, this technique significantly reduces the number of variables and constraints, especially those stemming from copy constraints. For instance, in circuits where multiple components share a common value or pattern, recognizing these cycles allows for a more compact representation, minimizing the solver's workload.

This new design and optimization strategy culminate in a significant leap forward for circuit analysis, promising considerable improvements in solver performance and analysis efficiency. The impact of these innovations is not just theoretical; they have been empirically validated to offer substantial enhancements over previous version.

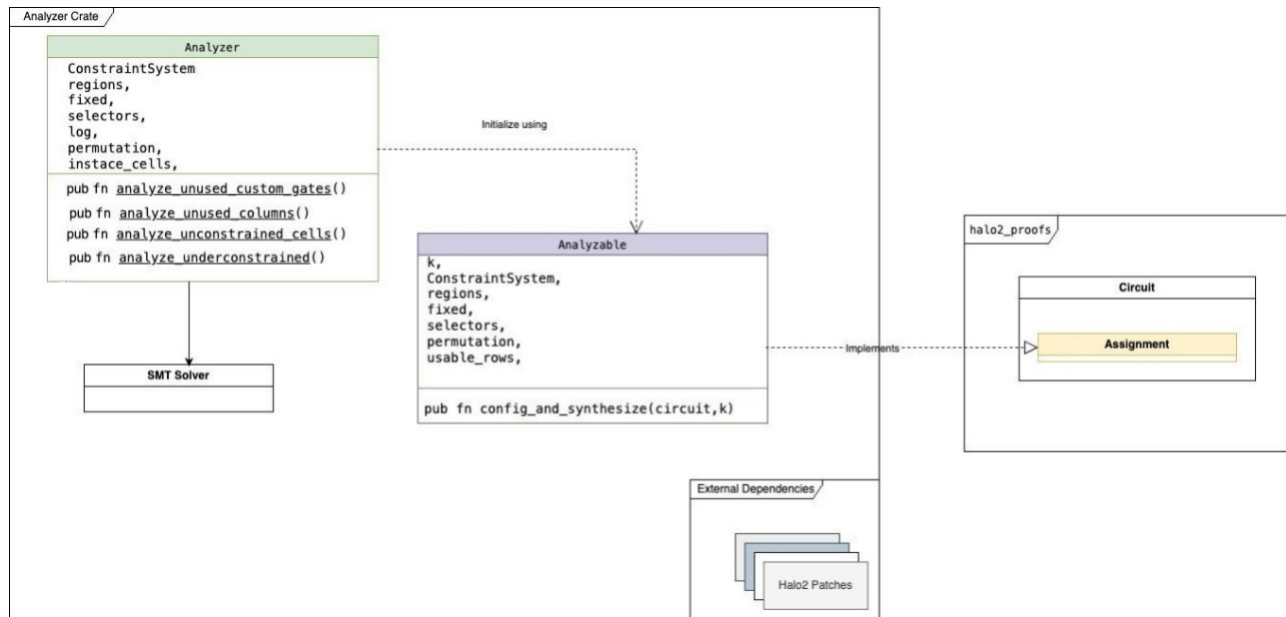


Figure 5.1 The redesigned halo2 Analyzer showcasing the integration of the Analyzable structure and cycle notion-based optimization.

The advancements encapsulated in the redesigned halo2 analyzer not only streamline the analysis process but also open new avenues for the development and verification of halo2 circuits.

5.4 Optimization Technique Based on Permutation Argument

The halo2 proving system utilizes a permutation argument to enforce the consistency of witness assignments by demonstrating that a certain permutation of variables, as dictated by the circuit's constraints, is maintained. A permutation in this context is defined as a one-to-one and onto mapping of a set onto itself, which can be uniquely factored into a composition of cycles.

5.4.1 Understanding Permutation and Cycles

A permutation is constructed with the goal that each subset of variables, constrained by equality, forms a cycle. This is conceptualized through cycle notation, where, for instance, a cycle $(a\ b\ c)$ denotes a mapping where a maps to b , b to c , and c back to a . The construction of such permutations allows for an efficient representation and verification of equality constraints within a circuit.

The algorithm for constructing the permutation starts with the identity permutation, where each element maps to itself. Upon introducing an equality constraint (e.g., $a \equiv b$), the algorithm checks if the involved elements are already part of the same cycle. If not, it merges the cycles containing these elements. This process ensures that elements within the same equality-constraint set are grouped into a single cycle, facilitating an optimized representation of the circuit's logical structure.

5.4.2 Permutation Schema in Halo2

The following schema represents a simplified view of the `Assembly` struct used in halo2 to organize permutation arguments within a circuit. This structure is pivotal in identifying cyclical relationships among variables for optimization purposes.

```
Assembly {
    // Columns represent the various types and indices of columns in the permutation.
    columns: [
```

```

        Column { index: 0, column_type: Advice },
        Column { index: 1, column_type: Advice },
        Column { index: 2, column_type: Advice },
        Column { index: 0, column_type: Instance }
    ],

    // Mapping defines the relationships between cells, essentially
    // detailing how variables are permuted across the circuit.
    mapping: [
        // Each tuple represents a mapping from one variable to another,
        ...
    ],

    // Sizes array stores the size of each cycle
    sizes: [
        ...
    ]
}

```

Each component of the **Assembly** struct plays a critical role in the permutation argument:

- **Columns:** Keeps the track of columns whose rows are constrained via copy constraints by their type (e.g., Advice, Fixed, Instance) and index.
- **Mapping:** Central to defining the permutation argument by specifying the exact relationships and transitions between cells.
- **Sizes:** The `sizes` array records the length of these cycles.

This schema underscores the intricacies of constructing permutation arguments in halo2 and highlights the potential for optimization by analyzing and simplifying the cyclical relationships among variables. By leveraging this structure, the proposed optimization technique effectively enhances the efficiency of the SMT solver, streamlining the analysis of halo2 circuits. Figure 5.2 shows how the matrices within the Assembly struct together represent the permutation cycles within a sample fibonacci circuit.

Each entry in Sizes matrix represents the size of the cycle that the corresponding cell belongs to. And the corresponding cell in the same indices in mapping points to the next cell in the cycle within the Mapping table.

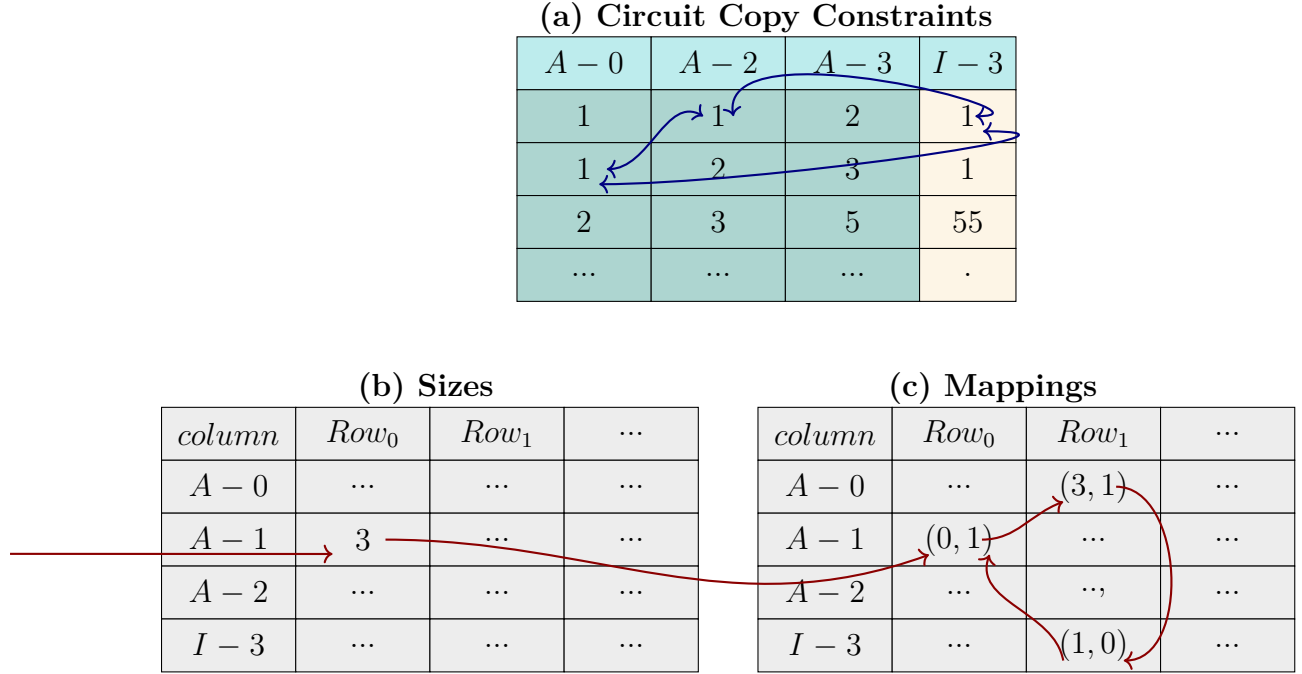


Figure 5.2 Tracking a sample cycle Via Assembly of a Circuit: (a) Copy constraints (b) Assembly.sizes Matrix (c) Assembly.mapping Matrix

5.4.3 Optimization Implications for the SMT Solver

The optimization technique developed for the halo2 analyzer employs an approach to handling permutation arguments, specifically tailored to enhance the performance of the SMT solver. The primary objective of this algorithm is to streamline the analysis of cryptographic circuits by minimizing the complexity inherent in verifying equality constraints among circuit variables.

The optimization process for identifying the cycles representatives to later propagate them within the SMT solver is outlined in the Algorithm 4.

This pseudocode abstractly describes the process of identifying cycles among variables constrained to be equal, identifying a cell within the cycle is the cycle representative. These representative cells are used in the SMT solver by substituting members of each cycle with a single representative in all the polynomials. Using a representative also allows us to remove all the copy constraints from the set of constraints in SMT solver. This technique streamlines the set of variables and constraints, facilitating a more efficient analysis. We prioritize adding an instance cell as the representative, as the analyzer has the option to analyze the circuit upon specific public inputs, which are of type instance, received from user.

Algorithm 4 Extract Permutations and Identify Cycles

```

1: Input: Assembly of the circuit
2: Output: CR // Cycle Representative
3: CR  $\leftarrow \emptyset$ 
4: for ith cycle in Assembly do
5:   CRi  $\leftarrow \emptyset$ 
6:   if cycle.Size > 1 then
7:     CycleSizei  $\leftarrow$  cyclei.Size
8:     while CycleSizei > 1 do
9:       Identify next copy constraint  $x \leftrightarrow y$ 
10:      CRi  $\leftarrow x$ 
11:      if y is Instance then
12:        CRi  $\leftarrow y$ 
13:      end if
14:      move to the next copy constraint
15:      CycleSizei.-
16:    end while
17:  end if
18:  CR = CR  $\cup$  CRi
19: end for
20: return CR

```

This methodology directly addresses the computational overhead associated with the permutation arguments required for validating equality constraints in the SMT solver. By effectively reducing the total number of variables to the count of unique cycles identified within the circuit, the algorithm significantly diminishes the solver’s workload. Each cycle’s representative acts as a stand-in for all variables within that cycle, thereby consolidating multiple equality constraints into a singular, more manageable form.

Such consolidation has profound implications for the efficiency of the SMT solver. It not only accelerates the verification process by curtailing the quantity of variables and constraints to be processed but also reduces the overall number of variables within polynomial constraints formed from gates. Consequently, this optimization leads to a more streamlined and computationally efficient analysis of halo2 circuits, underscoring the potential of cycle-based approaches in optimizing SMT solver operations within the context of cryptographic circuit analysis.

5.4.4 Practical Example: Fibonacci Circuit

The Fibonacci circuit serves as a practical example to demonstrate the application of our optimization technique. It’s designed using the halo2 proof system to compute Fibonacci

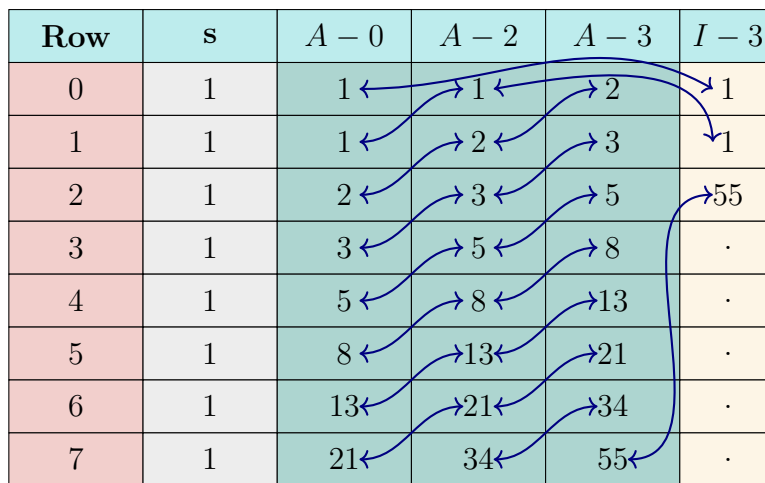
numbers, a sequence where each number is the sum of the two preceding ones. This circuit is a representation of how arithmetic operations can be embedded within a zero-knowledge-proof environment.

Circuit Configuration The circuit employs a `FibonacciConfig` structure to define its configuration, including:

- Advice columns (`col_a`, `col_b`, `col_c`) for storing current and previous Fibonacci numbers.
- A selector to activate the addition gate for computing the next number in the sequence.
- An instance column to expose the first initial fibonacci numbers and last computed Fibonacci number as public inputs.

Computation and Synthesis A `FibonacciChip` structure, parameterized over a field F , orchestrates the computation. It initializes the sequence with the first two numbers and iteratively computes subsequent numbers. The computation leverages a custom gate defined in the configuration to perform the addition operation $a + b = c$, where a and b represent the current and previous numbers, respectively, and c is the next number.

The circuit exposes the two first fibonacci elements and the last computed Fibonacci number as public inputs, allowing for its verification. This is a crucial for demonstrating the correctness of the computation within a zero-knowledge proof.



The diagram shows a table with 6 columns: Row, s, A-0, A-2, A-3, and I-3. The rows are indexed 0 to 7. The 's' column contains the value 1 for all rows. The 'A-0', 'A-2', and 'A-3' columns contain Fibonacci numbers: Row 0 has 1, 1, 2; Row 1 has 1, 2, 3; Row 2 has 2, 3, 5; Row 3 has 3, 5, 8; Row 4 has 5, 8, 13; Row 5 has 8, 13, 21; Row 6 has 13, 21, 34; Row 7 has 21, 34, 55. The 'I-3' column contains 1 for Row 0, 1 for Row 1, and a dot (.) for Rows 2 through 7. Blue arrows indicate a permutation mapping the values in the 'A-0', 'A-2', and 'A-3' columns to the 'I-3' column. Specifically, the value 1 from 'A-0' in Row 0 maps to 'I-3' in Row 0; the value 1 from 'A-2' in Row 0 maps to 'I-3' in Row 1; the value 2 from 'A-3' in Row 0 maps to 'I-3' in Row 2; the value 3 from 'A-3' in Row 1 maps to 'I-3' in Row 3; the value 5 from 'A-3' in Row 2 maps to 'I-3' in Row 4; the value 8 from 'A-3' in Row 3 maps to 'I-3' in Row 5; the value 13 from 'A-3' in Row 4 maps to 'I-3' in Row 6; the value 21 from 'A-3' in Row 5 maps to 'I-3' in Row 7; the value 34 from 'A-3' in Row 6 maps to 'I-3' in Row 7; and the value 55 from 'A-3' in Row 7 maps to 'I-3' in Row 7.

Row	s	A - 0	A - 2	A - 3	I - 3
0	1	1	1	2	1
1	1	1	2	3	1
2	1	2	3	5	55
3	1	3	5	8	.
4	1	5	8	13	.
5	1	8	13	21	.
6	1	13	21	34	.
7	1	21	34	55	.

Figure 5.3 Permutation argument for the Fibonacci circuit.

Extracting the permutation cycles of a Fibonacci circuit which calculates the 7th number of fibonacci, we will have the following representation of the cycles:

```

A-0-0->I-3-0->A-0-0
A-0-1->I-3-1->A-1-0->A-0-1
A-0-2->A-1-1->A-2-0->A-0-2
A-0-3->A-1-2->A-2-1->A-0-3
A-0-4->A-1-3->A-2-2->A-0-4
A-0-5->A-1-4->A-2-3->A-0-5
A-0-6->A-1-5->A-2-4->A-0-6
A-0-7->A-1-6->A-2-5->A-0-7
A-1-7->A-2-6->A-1-7
I-3-2->A-2-7->I-3-2

```

by employing algorithm 4 seven cycles are identified, and each cycle is assigned with a representative. For the above example the cycle representatives will be:

$$CR = \{CR_1 = I-3-0, CR_2 = I-3-1, CR_3 = A-0-2, CR_4 = A-0-3, \dots, CR_{10} = I-3-2\}$$

5.5 Evaluation

In the evaluation section, we conduct a comparative analysis of the two versions of the halo2 analyzer using a custom-designed Fibonacci circuit. This circuit serves as a benchmark to assess and demonstrate the performance benefits of the new static design and the cycle notion-based optimization technique incorporated into the halo2 analyzer.

5.5.1 Benchmark Setup

The benchmarking process was designed to evaluate the performance enhancements introduced in the new version of the halo2 analyzer. This evaluation specifically utilized a set of halo2 circuits embodying a Fibonacci sequence computation to serve as the testbed for our analysis. The Fibonacci circuit, chosen for its iterative computational pattern, was parameterized to generate sequences of varying lengths, corresponding to circuit sizes ranging from 5 to 34 rows. This range was selected to provide a comprehensive view of the analyzer's performance across circuits of increasing complexity and size.

Each circuit instance was constructed to compute the Fibonacci sequence by employing a series of addition operations, facilitated through advice columns and controlled by a selector

to ensure the correct sequence generation. This setup not only tested the analyzer’s ability to handle the fundamental arithmetic operations but also its efficiency in managing the circuit’s constraints and verifying its correctness.

The benchmarks were executed on both versions of the halo2 analyzer, recording the execution time for each circuit size. This direct comparison aimed to quantify the performance improvements attributed to the new static design and the cycle notion-based optimization technique. By focusing on execution times as the primary metric, we aim to demonstrate the tangible benefits of the optimizations introduced in the latest version of the analyzer.

5.5.2 Benchmark Results

The benchmarking process focused on evaluating the execution times required by both versions of the halo2 analyzer to analyze the Fibonacci circuit across varying sizes. The results, presented in Table 5.1, elucidate the performance improvements rendered by the new version of the analyzer.

Rows	Version 1 Time	Version 2 Time
5	341.694ms	291.537ms
8	440.775ms	338.120ms
13	1.333s	651.771ms
21	7.781s	2.210s
34	32.982s	11.041s

Table 5.1 Benchmark results comparing execution times of Version 1 and Version 2 of the halo2 analyzer on the Fibonacci circuits of various row sizes.

The benchmarking exercise reveals a pronounced reduction in analysis times with the implementation of the static design and cycle notion-based optimization in Version 2 of the halo2 analyzer, particularly as the circuit size increases. The optimized analyzer demonstrates a substantial efficiency gain, evidenced by the diminishing execution times relative to the circuit’s complexity and size.

5.6 Conclusion

In this section, we proposed a new design for the halo2 analyzer utilizing a custom implementation of the Assignment trait, facilitating static analysis of circuits through direct access to a novel data structure termed Assembly, which delineates the circuit’s copy constraints. These constraints are effectively represented through multiple matrices, enabling the encapsulation of cycles within cells. We introduced an optimization strategy predicated

on these cycles aimed at decreasing the number of SMT variables and constraints, which our benchmarks indicate yields a significant enhancement in execution time. For future works, further optimization could be achieved by advancing the handling of lookups in SMT solver, enhancing their efficiency in dealing with complex constraint scenarios. Additionally, the integration of preprocessing techniques such as abstract interpretation could offer a prelude to more rigorous analysis by approximating the circuit's behaviour, potentially streamlining the subsequent SMT-solving process. These avenues not only promise further reductions in computational overhead but also pave the way for more sophisticated analyses capable of handling increasingly intricate circuit designs.

CHAPTER 6 ABSTRACT INTERPRETATION FOR PLONKISH CIRCUITS

6.1 Introduction

As a result of recent research advancements, *zero-knowledge proof systems* have emerged as a key technology for various use cases. These proof systems have a number of desirable properties that allow a *prover* to convince a *verifier* that a certain statement is true without revealing any other information. These advancements have taken this class of cryptographic tools that have been known for years into the realm of practicality. As a result, these proof systems have found a large number of applications, including to scale blockchain networks.

General purpose blockchains like Ethereum [89] are secure but do not process many transactions per second. Through the use of so-called *layer two* networks, the transaction throughput of these ecosystems can be scaled up [101]. One way to do this is via so-called *zero-knowledge rollups*, each of which ties a blockchain network (the rollup, or layer two) to a general purpose blockchain like Ethereum (the layer one). The rollup network can execute transactions off-chain in a zero-knowledge proof system and post the proof on the underlying layer one network to be verified by a so-called smart contract [102].

This design works in because of two desirable properties of zero-knowledge proof systems: their proofs are *succinct* and the protocols are *non-interactive*. The fact that the proofs are *succinct* means that verifying the proof takes time poly-logarithmic of the statement being proved. The fact that the proof system is *non-interactive* means that prover can generate the proof in a single round and any verifier can check this proof at a later time and be convinced of its correctness. These systems are also *zero-knowledge* (ZK) if the verifier does not gain any information about what was proved beyond the fact that it is correct. Although some blockchain applications use the ZK property, many do not need this property. Zero-knowledge proof systems which have all three properties are often called a *zero-knowledge Succinct Non-interactive Argument of Knowledge* (or *zk-SNARK*). Examples of these proof systems include Groth16 [5], Halo [6], and PLONK [2].

The implementation of these proof systems is critical to their success and their applications. For example, at the time of writing, zero-knowledge rollups are responsible for managing more than \$3 billion USD on their networks according to the website L2Beat.com. Errors in these proof systems may allow malicious actors to submit incorrect proofs to steal funds from these networks or lock funds within them.

Preventing errors in these implementations is challenging due to the complexity of zk-SNARKs. zk-SNARKs often use a Domain-Specific Language (DSL) to allow developers to write their specific statement to be proved. This removes the complexity of these protocols and often allows automatic generation of verifier code (often targeting a blockchain smart contract language). The complexity of these proof systems often arise because they need to prove an NP relation and often do so using an *arithmetization* of the relation to be proved over a finite field. These arithmetizations are typically too complex to perform by hand for any useful relation, and the results are often too large to even be manually checked. Bugs in these systems can be introduced by the DSL compiler itself (see e.g. [103, 104]), or from the developer directly. As a result of this complexity, automated tools are crucial to ensuring that arithmetizations are correct.

In this work, we improve on one such tool so-called PLONKish arithmetizations. This arithmetization transforms a computation to a circuit comprising gates. The circuit is represented as a table of values which are constrained by an appropriate set of polynomials. PLONKish arithmetization is used in zk-SNARKs like Halo2 [14]. Halo2 has been used in a number of ZK rollups (e.g., Zircuit [9]) and there are a number of other proof systems based on PLONKish arithmetization. A more precise description of Halo2 is provided in Section 6.3.

We modify the abstract interpretation [21] techniques in the tool of [105]. Abstract interpretation can detect unused gates and columns which may indicate a bug (the developer may have intended to reference the gate or column, but did not). Abstract interpretation can also detect entries which are assigned but not constrained, which may indicate that the system is under-constrained. Under-constrained circuits are a particularly problematic concern for ZK proof systems (see e.g., [18, 19]). First, we add support for *copy constraints*, which enforce that particular entries in the table are expected to be the same. Second, we add support for *lookup functions* to the abstract interpretation. A lookup argument allows an entry to be valid only if it exists in a specified area of the table. This allows developers to specify valid ranges for particular values and to support pre-defined functions (as lookups may be over a tuple of table entries). We show that there are some lookup-implemented functions that can then also be used in the abstract interpretation itself. Supporting these features increases the accuracy of the abstract interpretation analysis.

Building upon the foundational concepts introduced in Chapters 4 and 5, this chapter extends the discussion on the automated analysis of halo2 circuits and introduces novel techniques for enhancing their reliability. Chapter 4 laid the groundwork by presenting methodologies for detecting under-constrained and over-constrained conditions in Halo2 circuits using SMT solvers. Chapter 5 further explored these concepts through the development of a static

analyzer optimized for performance via cycle-based notions.

Here, we advance these discussions by incorporating abstract interpretation techniques to address specific challenges associated with PLONKish circuits. While SMT solvers provide highly accurate results with no false negatives or false positives, their scalability remains a significant concern, particularly when analyzing complex cryptographic circuits at scale. In contrast, abstract interpretation offers a faster analysis but may yield false negatives or positives. This makes it ideally suited as a preliminary step in our analysis pipeline, efficiently narrowing down the areas of concern that require the more precise, albeit resource-intensive, scrutiny of SMT solvers.

The subsequent sections will detail the technical implementations of these enhancements and their implications for the security and efficiency of zero-knowledge proof systems in blockchain applications.

This paper is organized as follows. Preliminaries, including a detailed description of PLONKish arithmetization and Halo2, as well as the existing applications of abstract interpretation, are described in Section 6.3. Techniques for adding copy constraints and lookups are described in Section 6.4. Related work is described in Section 6.2 before the paper is concluded in Section 6.5.

6.2 Related Work

Correctness of ZK proof system implementations has been studied by others. As already mentioned, the authors of [105] built a tool which uses abstract interpretation for analysis as described in Section 6.3.3. However, the tool also has an SMT-based analysis for under-constrained circuits. Other applications of SMT-based approaches are given by Vella and Alt [61], and Pailoor et al. [17] but they target so-called *Rank-1 Constraint System* (R1CS) arithmetizations. This is an alternative arithmetization and their approaches are not directly compatible with PLONKish representations. Other automated tools like Ecne [15], and Picus [16] also exist for checking R1CS arithmetizations.

There is also other work towards the formalization of these systems and formally verifying them. Thomas [64] provides a specification language, Orbis, for zk-SNARK programming. This work provides a way to write Halo2 circuits but avoids discussion on checking their correctness afterwards. Chin et al. [67] describe a language for formally verifying ZK applications. The language aims to provide stronger guarantees for applications written in a custom ZK language, rather than a quick analysis of PLONKish/Halo2 circuits.

6.3 Preliminaries

In this section, we introduce PLONKish arithmetization (Section 6.3.1) before describing and outlining abstract interpretation (Section 6.3.2) and its prior use on PLONKish circuits (Section 6.3.3).

6.3.1 PLONKish Arithmetization

PLONKish arithmetization of functions or computations into a set of equations to be verified was originally introduced by PLONK [2]. There are several variations or extensions of PLONK like HyperPLONK [94], TurboPLONK [95], and UltraPLONK [96]. These versions have different features like the number of inputs or outputs to the circuit, combining PLONK with so-called *lookup* tables, and more; however, they all use the core ideas of PLONKish arithmetization. Halo2 is a popular PLONKish zero-knowledge proof systems which has been applied to a number of applications and was originally used by Zcash [14].

PLONKish arithmetizations transforms a function into a two-dimensional table and constraints over it. The rows and columns of the table are filled with values from a finite field and entry in the table at row i and column j is called a cell. There are three main column types: *instance* columns, which represent public inputs to the circuit; *advice* columns, which represent private witnesses (private inputs and intermediate values in the computation); and *fixed* columns, whose values are determined by the circuit definition. Constraints are applied to the table values as either a *gate constraint*, *equality constraint* (a.k.a. *permutation constraint* or a *copy constraint*), or a *lookup*. Gate constraints (or simply *gates*) define a set of constraints on cells of the table; these take the form of a polynomial defined on the constrained cells and are defined by the circuit designer. The cells in a gate equation can be of any type, and by default, this gate is active over all rows of the table.

To imply a gate is active on certain rows, *selectors* variables are used to activate and deactivate the rows for a gate. The selector is a column (of fixed type) with binary values in the table, and the gate polynomials are multiplied by the selector value in the current row; disabled gates are those in rows with a zero selector value. The cells in the gate are determined by the given column and row. This is implemented via relative references, which determine the column and the offset of the row from the row in which the selector is enabled. The gate constraints in combination with relative references and selectors shape disjoint regions over the circuit's table. Equality constraints construct a permutation by enforcing pairs of cells to be equal. These constraints form a set of disjoint cycles over the cells of the table, in which all the cells should be equal. Finally, a lookup constraint (or a lookup argument) enforces that

Table 6.1 An example of a Halo2 circuit’s table, simplified. The table has a number of rows that span selector (S_i), advice (A_i), lookup (L_i^j) or fixed (F_i) (in fact, there may be multiple different lookup functions L^j , which we group for convenience), and instance (I_i) columns. Selector column values are either 0 or 1 while the remaining columns can range over the entire finite field. The values of the Selector columns are set at circuit configuration and the lookup or fixed columns are public constants known to both the prover and the verifier. The advice columns are private and known only to the prover and entered when the statement is to be proved. Solid arrows indicate a copy constraint between the relevant cells of the table, namely, both ends of the arrow should have the same value when the entire table is evaluated. Dashed arrows represent lookups. The red cells may be empty or unassigned when the table is populated; not every cell is necessarily used.

row	S_0	A_0	A_1	A_2	F_0	L_0^0	L_1^0	L_2^0	I_0
1	1			$a_{2,1}$					
2	1	$a_{0,2}$	$a_{1,2}$	$a_{2,2}$					
3	0					$L_{0,3}^0$	$L_{1,3}^0$	$L_{2,3}^0$	
4	1	$a_{0,4}$	$a_{1,4}$	$a_{2,4}$					

a set of cell values should occur in the same row in a pre-defined sub-table of fixed columns (a subset of the fixed columns and the rows within them). Lookup constraints are a part of the circuit definition. PLONKish arithmetization requires the prover, given the table, to fill in the advice columns such that all polynomial constraints vanish (i.e., evaluate to zero). Oversimplifying, the structure of the constraints ensures that the prover must actually know the values to do this with all but negligible probability.

A simplified example PLONKish circuit table is shown in Table 6.1. The columns are arranged by type (selector, S ; advice, A_i ; fixed I and a lookup L^0 ; and instance I_i). The solid arrow between $a_{0,2}$ and $a_{2,1}$ enforces a copy constraint $a_{0,2} \rightarrow a_{2,1}$ that says these values should be equal. The dashed arrows between the entries in row 4 and the lookup entries on row 3 are a lookup: the values $a_{0,4}$, $a_{1,4}$, and $a_{2,4}$ should appear in some row of the columns L_0^0 , L_1^0 , and L_2^0 , in that order (but not necessarily row 3 as illustrated). Some parts of the table may not be used (indicated by red cells), as the construction typically uses table sizes that are the smallest power of 2 to fit the necessary rows. For large circuits, tables may have as many as 2^{26} rows in practice.

Halo2 has an implementation in Rust (used by Zcash and forked by others), which includes various traits that defines the components of a circuit (e.g., the assignments and layouts of

the table). This implementation can be modified to collect all the polynomial constraints within a circuit, the so-called **Assembly** which contains all the copy constraints, the details of the regions, and the enabled selectors within each region. In short, the implementation can be used to provide all of this information to a system that wishes to analyse Halo2 circuits.

6.3.2 Abstract Interpretation

We give a brief introduction to abstract interpretation as we use it in this paper. We use the abstraction function $\alpha(x)$ from a concrete set L to an abstract set L' . The concrete set L is the integers mod p where p is the order of the finite field used in Halo2 (e.g., for Zcash’s use of Halo2, it is the order of the finite field BN254 [106]). The abstract set is $L' = \{0, +, \perp\}$, where 0 is reserved for values known to be zero, $+$ is for values known to be integers (which can only be positive over the finite field), and $\perp$ is for values which are not known to be positive or zero, i.e., variables. The abstract set is represented in code as $L' = \{\text{Zero}, \text{NonZero}, \text{Variable}\}$. This simple abstract set is sufficient to determine if polynomials are identically zero by defining functions like `sum` and `multiply` over these domains. For example, the result of multiplying $\alpha(x) \cdot \alpha(y)$ where $\alpha(x) = 0$ and $\alpha(y) = \perp$ is 0. The justification for checking whether a polynomial is identically zero is described in the next section. Finally, we say that for $x, y \in L'$, $x \prec y$ if and only if $x = 0$ or both $x = +$ and $y = \perp$. The relation $x \prec y$ indicates that x is more concrete than y .

6.3.3 Prior use of Abstract Interpretation

We first describe the checks implemented in [105] which use abstract interpretation. Abstract interpretation is used to check for three things: *unused (custom) gates*, *unconstrained cells*, and *unused columns*. To check if there are unused (custom) gates, one needs to check that for every gate’s polynomials there exists a region in which it is not identically zero. Gates which are not used may be bugs as likely they were intended to be referenced. Similarly, for unconstrained cells, one needs to check that for every assigned cell in the region, it occurs in a polynomial which is not identically zero over this region. Instances of unconstrained cells typically indicate a bug as the system would be under-constrained. With these two checks, it is also possible to check for *unused columns*. To do this, one needs to check that every column occurs in some (not identically zero) polynomial. Like unused gates, unused columns may be a bug (perhaps the columns were meant to be referenced); however, they could simply enable an optimization by removing unused columns. In all cases, the relevant data is provided by a modified version of the relevant Rust trait.

These checks are quick but inaccurate, due to the nature of abstract interpretation. In

particular, the system may return that a polynomial is not identically zero when in fact it is; these are false negatives. As a result, the results need to be manually confirmed.

6.4 Improvements

In this section, we describe how to support copy constraints (Section 6.4.1) and lookup functions (Section 6.4.2).

6.4.1 Adding Support for Copy Constraints

Supporting copy constraints in the analysis requires ensuring that fixed values are propagated along the constraints. That is, if two table values $a_{i,j}$ and $a_{i',j'}$ are constrained to have the same abstract value; if either of the variable is known to be **NonZero** or **Zero**, that abstract value should apply to the other. Copy constraints can result in arbitrarily long cycles of variables being equal, and knowledge of the abstract value needs to be propagated to all of the table entries within the cycle. Table 6.1 shows an example of a table layout where copy constraints are indicated by arrows. In this example, there is a copy constraint between entry $a_{2,1}$ and $a_{0,2}$, so there is a cycle $a_{2,1} \rightarrow a_{0,2} \rightarrow a_{2,1}$.

Recall that some values are not known when the table is constructed: advice and instance columns values are added as the circuit is instantiated. These unknown values are given the **Variable** type. Algorithm 5 describes a method to propagate these values through a cycle. The key of the algorithm is to first identify the copy constraints related to a particular table entry and construct the cycles, before storing the most concrete value observed in the cycle through abstract interpretation. That is, if any element in the cycle is known to be **NonZero** or **Zero**, that value should be the one used for all entries in the cycle. If there are ever conflicting cases where one element in the cycle is known to be **NonZero** but another within the same cycle is known to be **Zero**, one can report that there is a bug in the circuits as both cannot be true simultaneously.

6.4.2 Adding Support for Lookups

Copy constraints allow values that map to 0 and + to propagate throughout the abstract interpretation as it is used to evaluate the polynomials constraining the table. However, some copy constraints actually come from copying values from a part of the table representing a lookup. That is, some copy constraints enforce that the prover computed values in order to satisfy a function described in a specified lookup. Suppose that we have a circuit with a number of lookups of the form $L^i(x_u, \dots, x_v)$. If L^i represents a function that returns a

Algorithm 5 Extract Cycle Representative Cells and Cycle Abstract Evaluations

```

1: Input: Assembly of the circuit, fixed columns' assignment
2: Output: CAE // Cycle Abstract Evaluation
3: CAE  $\leftarrow \perp$ 
4: for cycle in Assembly do
5:   if cycle.Size > 1 then
6:     CycleSize  $\leftarrow$  cycle.Size
7:     while CycleSize > 1 do
8:       Identify next copy constraint  $x \leftrightarrow y$ 
9:       if  $\alpha(x) \prec$  CAE then
10:        CAE  $\leftarrow \alpha(x)$ 
11:       end if
12:       if  $\alpha(y) \prec$  CAE then
13:        CAE  $\leftarrow \alpha(y)$ 
14:       end if move to the next pair
15:     CycleSize-
16:   end while
17: end if
18: end for
19: return CAE

```

single value, the entry x_v is taken to be the function's output. In general, a lookup may return a t -tuple of results, in which case we use the last t values as the output. Moreover, a lookup need not be a function: there may be multiple rows which the same “inputs” but different outputs, if the lookup is not deterministic or the “output” is allowed to be within a range of values. For simplicity, we assume that we have lookups which represent functions that return a single output value y .

In Table 6.1, a simplified example of this is shown by the dashed copy constraints $a_{0,2} \rightarrow l_{0,3}^0$, $a_{1,2} \rightarrow l_{1,3}^0$, and $a_{2,2} \rightarrow l_{2,3}^0$. These constraints may be enforcing that a given lookup function $L^0(x_0, x_1, x_2)$ is being executed correctly: the table will be instantiated with concrete values $l_{0,3}^0$, $l_{1,3}^0$, and $l_{2,3}^0$ such that $L^0(l_{0,3}^0, l_{1,3}^0, l_{2,3}^0)$ is in the rows of L^i . This way, the prover cannot “cheat” for values $\{a_{0,2}, a_{1,2}, a_{2,2}\}$: these values must map to the row with values $\{l_{0,3}^0, l_{1,3}^0, l_{2,3}^0\}$ because of the copy constraints.

In general, the row for the lookup is not known prior to the population of the advice columns (unless there is only one such row). Instead, the actual constraints enforce that $\{a_{0,2}, a_{1,2}, a_{2,2}\}$ maps to $\{l_{0,k}^0, l_{1,k}^0, l_{2,k}^0\}$ for some k such that the columns of L^0 are defined; this allows the prover to use which ever values are needed in the advice columns, so that some valid execution of L^0 is represented. If no row is possible, the prover will be unable to satisfy the constraints (and therefore be unable to generate a valid proof).

If there is only one row for the lookup (namely, one acceptable input and output for the lookup), copy constraints can be used. In such a case, the values from the only valid lookup can be evaluated with the abstraction function α and the results can be propagated into the advice columns (and through any cycles created by copy constraints).

However, as most interesting lookup arguments have multiple rows, the index of the row is not known. In this case, the approach of value propagation through copy constraints is not valid: we do not know which values to propagate. One way to overcome this would be to create a copy of the constraint system for every valid row in every lookup. However, given that tables are often very large (and so the lookups may also be very large), this would create a very large number of instances to check.

We take another approach and try to apply abstract interpretation to the lookup function itself. For example, if it is known that a lookup $L^i(x_0, x_1, x_2)$ is the bitwise AND operation, the lookup can be interpreted over the abstract domain. The result of the operation for 3-bit numbers is shown in Figure 6.1a, which shows that if either argument to the function is zero, the result is too. This example may appear contrived, but bitwise functions are actually difficult to directly compute using integers over finite fields, and are often included as lookup functions.

We now explain this idea in general. For each lookup $L^i(x_u, \dots, x_v = y)$, the idea is to create an abstract evaluation function based on the matching abstract type of $\alpha(x_u)$ for $u \in \{0, \dots, v-1\}$ and the abstract type of $\alpha(x_v) = \alpha(y)$. Given the abstract domain values, there are at most $(|L'| + 1)^{v-1} = 4^{v-1}$ patterns to match within an abstract evaluation function: one for each of the elements in L' , and one more representing the placeholder expression “_” which matches any type. Patterns that result in the outcome being **Zero** are particularly useful. This is because the analysis is checking to see if polynomials are identically zero and zeros are the most concrete element in the abstract set L' . The matching patterns may need to be sorted so that the results that map to useful outcomes (i.e., **Zero**) are presented prior to ones that may still result in a match but match to something more general. An example abstract evaluation function for the lookup of the bitwise AND of two 3-bit integers is shown in Figure 6.1b.

The creation of abstract evaluation functions rules is a challenge. In this work, we assume that such a function is hand-written and hard-coded into the abstract interpretation engine. That is, if the circuit designer knows that a lookup is amenable to a custom abstract interpretation rule like the one in Figure 6.1b, they can add such a abstract evaluation function. It is an interesting open problem if these evaluation functions (or simplified versions of them) can be automatically built given the circuit representation of the lookups, perhaps as a pre-

processing step for the analysis itself.

For each lookup L^i for which an abstract evaluation function L_i can be written, we can use the rule during the abstract interpretation itself. As an illustrating example, consider again the lookup in Table 6.1 and assume that L^0 is the bitwise AND function show in Figure 6.1a. After initial values are set and propagated via the copy constraints, if either $\alpha(a_{0,4}) = 0$ or $\alpha(a_{1,4}) = 0$, then the corresponding abstract evaluation function L_0 will return $\alpha(a_{2,4}) = 0$ as well. Then, we can propagate $\alpha(a_{2,4}) = 0$ through any copy constraints and continue, or repeat our analysis. In particular, if the abstract evaluation function’s output is in a copy constraint and is more concrete than any previously computed ones, we propagate that value through the cycle of copy constraints and reevaluate any polynomials that use any variables in the cycle.

6.5 Conclusion

We presented methods to support key features of PLONKish arithmetizations in abstract interpretation-based analysis of circuits. We describe methods to support copy constraints and lookup arguments. Copy constraints improve the accuracy of the analysis by propagating known values across the constraint system. Abstract interpretation can also be used to analyse custom lookup-implemented functions in situations where those functions have nice properties.

We presented a motivating example that required the property that if any argument to the function is zero, the output is as well. However, it would be interesting to determine if other properties can be used to build helpful abstract evaluation functions and if any such functions can be automatically built from the circuit source code. Our analysis uses an abstraction function which has only three elements in the abstract domain; it would also be interesting to see if there are more advanced abstraction functions which can leverage other useful abstract evaluation functions.

Abstract interpretation is an excellent tool for the analysis of PLONKish circuits. Although it may report false negatives, its efficiency means that it is able to quickly check entire circuits and the relevant constraint systems, which may have as many as 2^{26} entries. In turn, we have taken steps towards automatically analyzing PLONKish circuits and safeguarding the applications that use them.

x	y AND	0	+						
		000	001	010	011	100	101	110	111
0	000	000	000	000	000	000	000	000	000
+	001	000	001	000	001	000	001	000	001
	010	000	000	010	010	000	000	010	010
	011	000	001	010	011	000	001	010	011
	100	000	000	000	000	100	100	100	100
	101	000	001	000	001	100	101	100	101
	110	000	000	010	010	100	100	110	110
	111	000	001	010	011	100	101	110	111

(a) The bitwise AND lookup function ($x \& y$) for 3-bit numbers. There is no clear rule for $+$ & $+$, $\perp$ & $+$, $+$ & $\perp$, or $\perp$ & $\perp$.

```

1 Expression::AndLookup(left, right) => {
2   let res1 = eval_abstract(left, ...);
3   let res2 = eval_abstract(right, ...);
4   match (res1, res2) {
5     (AbsResult::Zero, _) => AbsResult::Zero,
6     (_, AbsResult::Zero) => AbsResult::Zero,
7     (AbsResult::Variable, _) => AbsResult::Variable, // could be anything
8     (_, AbsResult::Variable) => AbsResult::Variable, // could be anything
9   }
10 }
```

Listing (6.1) AnalyticalShape Struct

(b) An example of evaluating the “bitwise AND” lookup function using abstract interpretation.

Figure 6.1 The evaluation table for bitwise AND on 3-bit numbers (a) and how to abstractly evaluate it (b).

CHAPTER 7 BLOCKCHAIN-BASED ROLE-CENTRIC ACCESS CONTROL AS A SERVICE

7.1 Blockchain-Based Role-Centric Access Control as a Service

In this chapter, we first define the concepts on which the access control management system is built, then examine the various possible designs for implementing such a system and the effect of selecting each of these designs from the perspective of the users, the administrators, and the overall system security, and finally provide details of the overall architecture of our platform. The depicted architecture in Figure 7.1 illustrates the suggested platform for Access Control as a Service. Within this platform, users interact with the system to register and request access to various resources. Additionally, owners of smart contracts interact with the access control system to establish the service, define roles, grant roles to users, define attributes, assign attributes to users, assign attributes to resources, and define policies based on roles. Furthermore, access to roles is limited by implementing attribute-based policies on said roles.

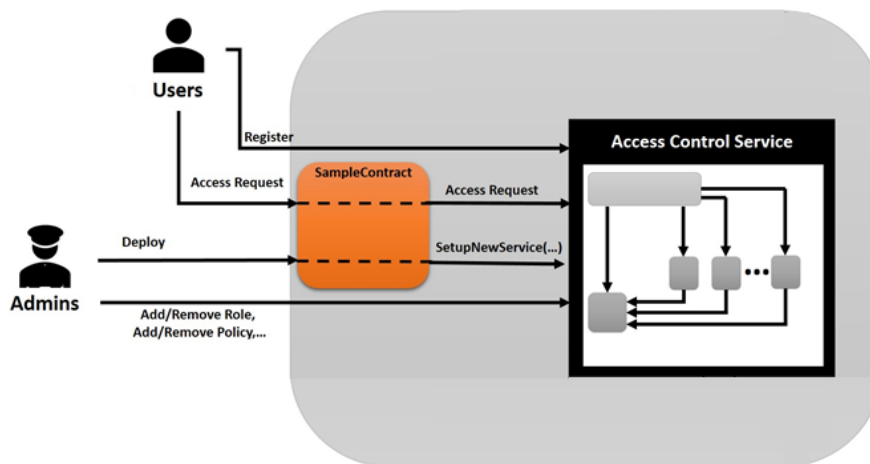


Figure 7.1 Access Control as a Service.

This chapter begins by presenting a comprehensive definition of the fundamental concepts that form the foundation of the access control management system. Subsequently, an exploration of the different potential designs for implementing this system is conducted, along

with an analysis of the implications associated with selecting each design from the viewpoints of the users, administrators, and overall system security. Lastly, a detailed overview of the overall architecture of our platform is provided.

To facilitate an explanation of the access control management ideas that will be covered in subsequent sections, a sample smart contract with a simplified structure is presented Listing 7.1.

```

1 contract Storage {
2
3     uint256 number;
4
5     function store(uint256 num) public {
6         number = num;
7     }
8
9     function retrieve() public view returns (uint256){
10         return number;
11     }
12 }

```

Listing 7.1 A sample Storage contract

In Ethereum, each smart contract is identified with its distinct addresses. In addition, Ethereum employs an Application Binary Interface Specification for enabling communication, encompassing both function calls originating from externally owned accounts and function calls originating from other contracts. According to this standard, every function has a unique identifier that can be determined from the hash of its signature. As an illustration, it can be observed that the identifier (ID) associated with the function **store(uint256)** corresponds to the initial four bytes of the hash value generated by applying the Keccak hash function to the function signature **store(uint256)**. The hexadecimal representation of the function signature for the method **store(uint256)** is **0x6057361d**.

The proposed architecture will be implemented using the established standard of smart contracts in Ethereum.

7.1.1 Role-Centric Access Control Model for Smart Contracts

This section formally defines different concepts of a Role-Centric Access Control Model adopted to manage smart contract requirements. These concepts include users, resources,

roles, attributes, and policies. A smart contract is shown in Listing 7.1 to understand these concepts better.

- **Resources:** In a role-centric access control architecture, the resources are considered entities that require users to be protected from unauthorized access. In the context of implementing access control management for smart contracts, resources refer to the functions within the smart contracts that are identifiable by their respective function IDs. Resources can be defined as a collection of function IDs.

$$\mathcal{F} = \{\mathcal{F}_i \mid \mathcal{F}_i \text{ is a function ID}\} \quad (7.1)$$

For example, for the contract shown in Listing 7.1, the resources set includes two function IDs of the **store(uint256)** and **retrieve()**, which will be unique in combination with the contract address.

- **Users:** In Ethereum, individual users are identified by their unique account addresses. The address serves as the primary identifier for users. According to this, users can be defined as a collection of addresses intended to interact with a decentralized application (DApp) by initiating transactions that result in specific function calls.

$$\mathcal{U} = \{\mathcal{U}_i \mid \mathcal{U}_i \text{ is a user}\} \quad (7.2)$$

- **Roles** In a role-centric access control management model, roles are defined based on some common characteristics that a group of users share. Roles are used to restrict access of the users to resources, by defining role-based policies. We can define roles as a set of unique role names:

$$\mathcal{R} = \{\mathcal{R}_i \mid \mathcal{R}_i \text{ is a role}\} \quad (7.3)$$

The relationship between users and roles can be defined as:

$$\{\mathcal{U}_i \rightarrow \mathcal{R}_j \mid \mathcal{U}_i \text{ is a user which has } \mathcal{R}_j\} \quad (7.4)$$

It is worth mentioning that a user can have multiple roles.

- **Attributes** Attributes are more specific characteristics of users. Users can get assigned to these attributes, and attribute-based policies are defined to employ extra restrictions on the role's access level. We can define attributes as a set of attribute names:

$$\mathcal{A} = \{\mathcal{A}_i \mid \mathcal{A}_i \text{ is an attribute}\} \quad (7.5)$$

Our model defines policies by using attributes as (**attribute, value**) pairs. The users are assigned attributes, and regarding value for the attribute is determined:

$$\{\mathcal{U}_i \rightarrow (\mathcal{A}_j, \mathcal{V}_k) \mid \mathcal{U}_i \text{ is a user which has value } \mathcal{V}_k \text{ for attribute } \mathcal{A}_j\} \quad (7.6)$$

For example, assume that the system admin defines an attribute **Company** to control the access control on different functionalities based on the company users work in. This attribute can be assigned to a user with address **0x71..6F**, and its value can be set to **C1**, meaning that the user with address **0x71..6F** works in the company **C1**.

- **Policies** Having the users, roles, resources, and attributes, we can define two types of role-based and attribute-based policies. For simplicity, assume that there is a Boolean variable **ACC** to show that the policy is defined to approve or deny access.
 - **Role-Based Policies** These policies define rules on roles and resources: which role has access to which resource. We can define these rules for each resource. We define policies for each resource as follows:

$$\mathcal{RBAC}_m = \{\mathcal{F}_i \rightarrow (\mathcal{R}_j \rightarrow \mathcal{ACC}_k)\} \quad (7.7)$$

$$\mathcal{ACC}_k = \begin{cases} \text{True} & \text{if role } \mathcal{R}_j \text{ has access to resource } \mathcal{F}_i, \\ \text{False} & \text{otherwise.} \end{cases}$$

For example, assume that a user with address **0x71..6F** has role **BUYER**. The admin can define a role-based policy **rbac** that gives users with role **BUYER** access to function **store(uint256)** with ID **0x6057361d**; enabling the mentioned user to execute function **store(uint256)**.

- **Attribute-Based policies** To restrict the access on roles, attribute-based policies can be defined for each role-based policy. These policies can be defined as:

$$\mathcal{ABAC}_n = \{\mathcal{RBAC}_m \rightarrow ((\mathcal{A}_i, \mathcal{V}_j) \rightarrow \mathcal{ACC}_k)\} \quad (7.8)$$

To explain these policies, consider the role-based policy explained before. Assume that the admin decides to restrict users' access with role **BUYER** based on their **Company** attribute. For example, all users with role **BUYER** have access to execute function **store(uint256)** except those which have their company attribute set to **C1**. Based on the formula (8), admin will be able to define an attribute-based policy on previously defined **rbac** and set the pair of (**attribute, value**) as (**COMPANY, C1**) with **ACC** variable false. during the policy control process users are checked against role-based policies to check if their role has access to a resource. If they have access, the role-based policy that gives them this access is checked to control if there are extra attribute-based policies defined for it. Which in this example means that, although user **0x71..6F** has role **BUYER**, and **rbac** gives users with role **BUYER** access to function **store(uint256)**, since the user has attribute **Company** set to **C1**, will not be able to execute this function.

7.1.2 Privacy-Preserving Access Control using Blockchain

Users' Privacy is one of the most controversial subjects about storing data on the blockchain. There is a common belief about blockchain, which claims users' privacy is protected using pseudonymity. In simple words, it means that the users do not connect to the blockchain with their real identities. In Ethereum, users need an account address to send their transactions to the network. Still, a user can create unlimited new addresses and even send each transaction using a new address. However, active studies have proved this pseudonymity can be broken by the deanonymization attacks [107]. The Zerocash was a fork of Bitcoin that proposed a solution to address its privacy problems [108]. But their solution, which uses zero-knowledge proof for anonymizing payments, is only efficient for payment systems. In addition, a growing number of DApps require to store their users' real identities for different purposes.

One of the characteristics of blockchain is its transparency, meaning that everything stored on blockchain is visible to all. In our study, we must store users' attributes on the blockchain. In addition, the policies are stored on blockchain, too. The transparency in the blockchain is a two-edged sword: while it makes the access control policy definitions auditable for all users, it threatens their privacy, too.

Let's explain these concepts in more detail with an example: a system admin defines a role **BORROWER**. The users are assigned to this role. A policy is defined for this role, granting users access to the Borrow function with the role **BORROWER** role. The admin also decides to restrict the access of **BORROWER** role with two attribute-based rules: the first rule is that users whose location attribute is L_1 are allowed to execute the Borrow function, the second rule is that, users whom their Department attribute is equal to D_1 are not allowed to execute the Borrow function.

A centralized system stores user attributes and policies in a central database. Users may have no access to the attributes defined for them and the policies. Since the system is not transparent, an admin can define policies without exposing them to the users. This causes a lack of auditability of access control management rules for users.

A decentralized platform like a public blockchain solves this problem. Everything on the blockchain is transparent and auditable. All the user attributes and policies are stored on blockchain. Any changes to the policies by the admin are visible to all users. The problem here is that the privacy of users is at risk.

Different solutions are proposed to address this issue. One solution is to store encrypted user attributes instead of plain text. But there is still a risk for privacy: assume there is a user $U1$ whose location attribute is equal to L_1 . The transparency of transactions on the blockchain means that all users' activities are visible to everyone. Assume that a transaction is stored on the blockchain, showing that user $U1$ has successfully executed the Borrow function. Since there is a policy which shows only users who for them $Location = L_1$ are allowed to execute the Borrow function, some external witness will conclude that the Location attribute of $U1$ is L_1 . This also shows that the Department attribute of $U1$ is not D_1 . This scenario exemplifies a well-known privacy issue in access control systems known as the inference problem [109]. Addressing such inference attacks, particularly in blockchain-based access control systems where transactions are publicly accessible, presents an intriguing avenue for research.

Another solution to this problem is to store encrypted policies on the blockchain, which, due to lack of transparency, eliminates auditability.

Authors of [84] proposed the PPDAC Model to enhance the privacy of the FairAccess platform [76], their previous access control framework for IoT on blockchain. FairAccess uses authorization tokens, but since the policies are stored on the public blockchain, there is the possibility that the privacy of users is at risk. Their PPDAC model uses CP-ABE to store encrypted policies on the blockchain. This model is privacy-preserving, but hiding the policies will sacrifice auditability, as we mentioned before.

As a middle-ground solution, we will only encrypt the attributes of the users and store the policies in Plain-Text mode. By this, the auditability of policies is preserved, and the users' privacy is preserved to some extent, providing a trade-off between privacy and auditability. However, it's important to acknowledge that our proposed privacy protection mechanism, while offering a certain level of privacy, has limitations regarding resistance against hash function guess attacks.

Attribute Encryption

To protect users' privacy, their encrypted attributes are stored on smart contracts. The *Keccak* – 256 hash function should encrypt values related to users' assigned attributes before sending them to the smart contract through a transaction. While *Keccak* – 256 is a robust cryptography hash function, it should be acknowledged that it does not provide full-proof protection against all kinds of attacks. Specifically, given enough time and computational resources, an attacker could potentially guess the input that resulted in a given hash. To further strengthen the security, a future direction could involve employing a technique such as salting, where a unique, random value is added to each attribute before hashing. This would make guessing the original attribute values more difficult even if the attacker knew the hash and the salt.

Assume that there is a policy defined by admin which grants access to the resource f_1 to the role R_1 . Admin decides to restrict this grant for users with role R_1 . The restriction is that only users whose Attribute A_1 equals to value v_1 will have access to f_1 .

As shown in Figure 7.2, the admin stores a constraint as $(A_1 == v_1)$. To make this policy auditable, we store the policy P_1 as " $A == v$ " in plain-text format. On the other side, we store the user's encrypted information. For example, for a user U_1 , the value of attribute A_1 is v' . Instead of simply storing v' on-chain, v' should be encrypted via *Keccak* – 256 hash function:

$$v'_{enc} = keccak256(v') \quad (7.9)$$

and sent via a transaction to the blockchain.

Once the user requests access to the resource R_1 , the value defined in the policy for the related attribute is encrypted and compared to v'_{enc} :

$$v_{enc} = keccak256(v) \quad (7.10)$$

If the user's attribute value satisfies the constraint, v_{enc} should be equal to v'_{enc} .

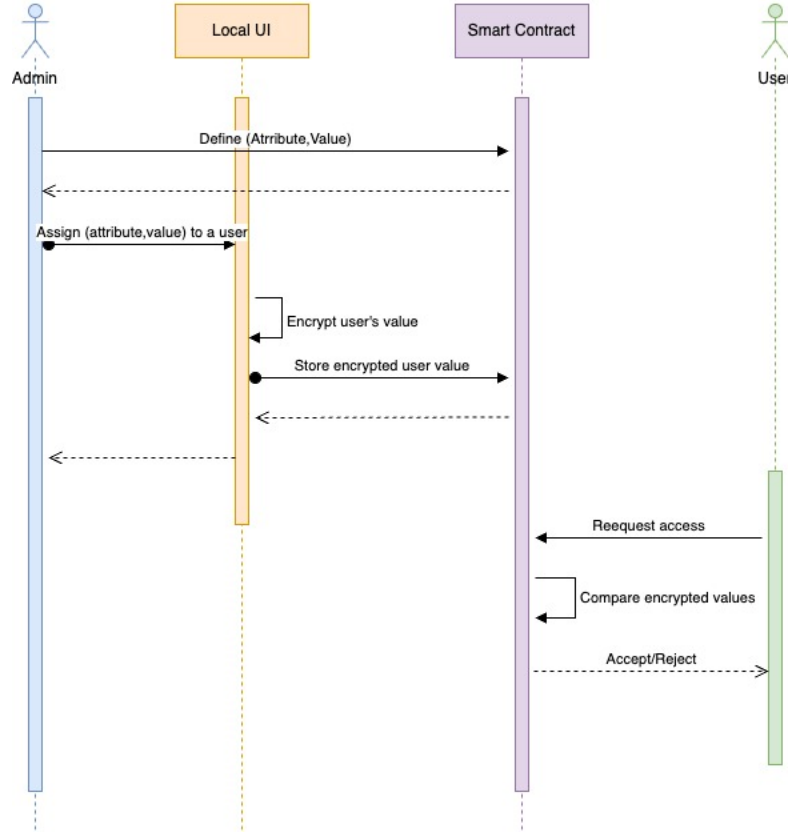


Figure 7.2 User's attributes encryption and validation

User Activities on the Blockchain

A significant point to note is that user activity records on the blockchain are not protected in our current model. The transparency of transactions on the blockchain, an inherent feature of this technology, does pose a privacy concern as all user transactions are publicly visible. This is an ongoing trade-off in blockchain between transparency and privacy. Future work could explore integrating privacy-focused techniques such as zero-knowledge proofs or secure multi-party computation. However, while offering enhanced privacy, these solutions introduce challenges, such as increased complexity and computational costs. A more in-depth exploration of these methods could further the field and provide more robust privacy-preserving mechanisms in blockchain-based access control systems.

7.1.3 Design Alternatives

In this part, we first introduce the deciding parameters that different design alternatives are compared based on them. Then, explore the different possible designs and their impact on these parameters.

- Gas Optimization is an important parameter in choosing a design pattern for a blockchain-based platform. Normally, gas consumption is important in two aspects: 1. Deployment cost, and 2. Transaction cost.
 - Deployment Cost

The cost of deploying a platform is the amount of gas consumed for a platform to get deployed on the blockchain. It depends on different parameters like code size and code complexity.
 - Transaction Cost

Upon deploying a DApp, users interact with the deployed DApp through transactions. Consequently, the cost of executing these transactions becomes important for users.

In our platform, there are a few specific considerations about gas consumption.

1. The Access Control Management logic is intended to be a generic implementation that does not vary across DApps with different business logic. It means that the ideal system design would be the one that does not deploy the whole access control management implementation for each new DApp and uses a previously deployed implementation.
 2. The users who interact with an access control management platform can be categorized into two main groups: DApp users and access control management admins. Thus, we can examine the effect of choosing different design patterns based on the transaction cost for admins and users.
- Decoupling refers to the modular separation of the main business logic and the access control management of a DApp.

It is worth mentioning that the following discussions about design decisions are about the theoretical aspect of deploying a general contract and might not apply to all smart contracts due to the limitations on the contract size that can be deployed on the Ethereum blockchain.

Single Contract

A simple design solution implements the whole business logic and access control management in the same contract. This design method leads to high dependency between access control and the business logic and increased complexity of the DApp. This complexity can be the source of many security vulnerabilities in the system. Considering the above-mentioned parameters, being strongly coupled is a negative point for choosing this solution. Deployment cost is high since the whole access control management logic is implemented and deployed. The transaction cost is low for both users and admins since the whole logic is deployed in the same address, and there are no external calls.

Factory Design Pattern

The factory design pattern for smart contracts has a similar definition as it has in object-oriented programming. It can be considered with the same concepts adapted to a contract-oriented environment. This design pattern is ideal when a standard implementation contains data storage and functionality, like Role-Centric access control management, which other contracts can use. In this pattern, there is a Factory contract and the implementation of the standard logic. The Factory is responsible for the deployment of new contracts using standard implementation. Regarding our deciding parameters, it can be said that the main advantage of this method is the total decoupling of business logic and access control management. This is an ideal design to help prevent vulnerabilities arising from the over-complexity of the contracts.

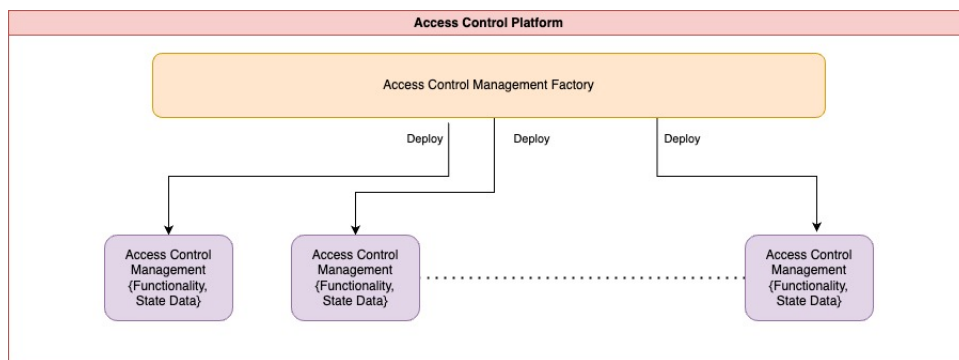


Figure 7.3 Factory Design Pattern

The discussion about gas optimization is more complex here. The deployment cost is as high as the previous method since there is no need to implement the access control management

from scratch, but for each new DApp, the whole implementation should be deployed in a new address. Each DApp has its specific access control management contract deployed on blockchain. From the transaction cost point of view, every call to a function that requires an access control check will perform an external call to the DApp's access control management contract address. This imposes an extra cost to users for each function call. There is no extra cost for admins to interact with access control management functionalities since the administrative interaction can be performed directly with the access control management address.

Minimal Proxy Design Pattern

Also known as Clone Factory Design Pattern. This design pattern is one of the Ethereum Improvement Proposals. It is a standard which defines a basic bytecode implementation that delegates all calls to a known, fixed address to immutably clone contract functionality [110]. There is a factory and an implementation similar to the factory design pattern in this design pattern. The implantation is deployed once, and all the new instances will be a proxy contract that delegates call to that initially deployed implementation. This Significantly reduces the deployment cost since the proxy contract is a minimal contract with a low gas cost. The Decoupling of access control and business logic is as high as the factory design pattern. The transaction cost for users is the same as the factory design pattern and still higher than the single contract solution because of the external calls. The administrative transaction cost is low.

Metrics	Single Contract	Factory	Minimal Proxy
Admin transaction cost	✓	✓	✓
User transaction cost	✓	-	-
Deployment Cost	-	-	✓
Decoupling	-	✓	✓

7.1.4 Proposed Design Solution

The heart of our access control as a service system is a main contract that handles user, role, and attribute management, policy definitions, and access requests. There are two possible designs for this contract. This contract needs to manage the address of all contracts that use its services, and for each address, manage users, roles, attributes, and policies. Implementing all the functionality in one contract is possible, but complex data structures will be required to manage related huge and complicated data. There is another solution that uses a factory

design pattern. The Main Contract uses the factory design pattern and deploys a child contract for each access control service request. For example, if there are two contracts, **C1** and **C2**, which want to use the access control service, the main contract deploys a child contract for each of them. This helps to keep the system simple, and as discussed before, less complexity leads to less vulnerability and erroneous coding. The only concern about the factory design pattern is the extra cost of deploying a new contract for every DApp. To solve this problem, we use the Minimal Proxy Design Pattern, which uses the Proxy design pattern which was explained in 7.1.3. The idea is that while all the child contracts have the same bytecodes to be deployed, we will pay the gas cost of contract deployment for the first child contract. Using the clone factory, besides reducing the complexity of the main contract, should result in a more gas-efficient implementation of the system.

Here, for simplicity, we consider the whole access control logic deployed once within one contract. But as we will discuss in the next sections, we have broken down this logic into multiple contracts, which are integrated and deployed together once.

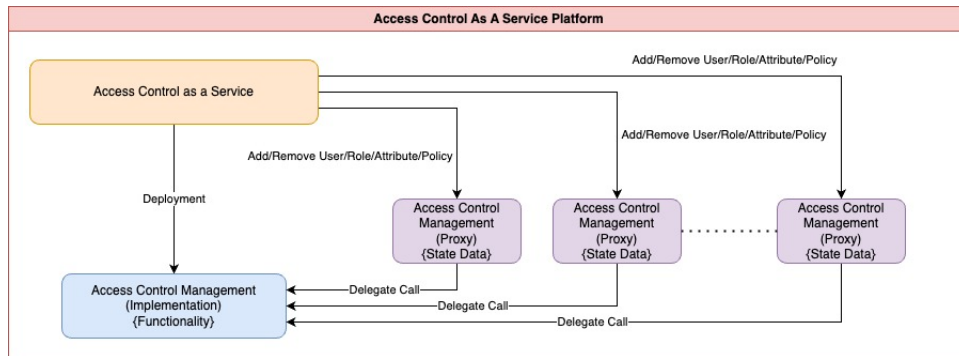


Figure 7.4 Proposed Design

7.1.5 Implementation: Ethereum-Based Role-Centric Access Control as a Service

The primary service provider nature of our Role-Centric Access Control as a Service on Ethereum revolves around the `IAccessControlAsService` contract. This contract is essential in managing services and implementing access control mechanisms inside the system, facilitating a comprehensive and efficient interface for contractual operations. The system adeptly integrates Role-Based Access Control (RBAC) with Attribute-Based Access Control (ABAC),

providing a precise access control mechanism and facilitating enhanced management functionalities.

Three main implementation contracts will be deployed on each service request received from a contract.

- `AccessControlManagement`:
- `ABACAccessControlManagement`
- `RBACAccessControlManagement`
- `StateAccessControl`

The `AccessControlManagement` which handles user management, role management, RBAC policy definitions, and static ABAC policy definitions. The other implementation is `StateAccessControl` for managing dynamic state-based ABAC policies. The `NewService()` function initiates a new service for the specified contract address and designates its corresponding administrator. The function creates a clone of the master contract, acting as the service provider and establishing a link with the original contract. Alongside, an instance of the `stateAccessControlmaster` contract is cloned, ensuring the management of dynamic state-based access control policies. Service lists are updated, ensuring accurate tracking.

The `accessControl()` function performs access control checks for each contract function invocation, processing the payload to extract the function signature and evaluating the sender's access rights. To improve system interactivity, the `userRegister()` function facilitates registering new users to a particular contract's service. Role management within this system is streamlined via functions such as `addRole()`, `grantRole()`, and `revokeRole()`, allowing service administrators to manage roles within a contract's service effectively. Additionally, administrators can use the `addResource()` function to augment the system's resources and the `addPolicy()` function to create RBAC policies associating a role with a resource and action type.

The `addAttribute()` function enables the addition of new attributes to a contract's service. These attributes are utilized in ABAC to define supplementary conditions for access control. The `addRBACABACs()` and `addUserABACs()` functions further extend the system's capabilities by allowing the service administrator to add condition-based policies for roles, thus unifying RBAC and ABAC principles. These policies can be based on user attributes or the return values of other contract function calls. One notable feature of this system is the `stateAccessControl` component, which facilitates dynamic, state-based access control

policies. The utilization of a state-based method offers enhanced flexibility, enabling the system to respond effectively to modifications in its state. Additionally, this technique enhances security by granting context-aware rules to give permissions, resulting in finer granularity over the control of contract interactions.

In summary, the Ethereum-Based Role-Centric Access Control as a Service, centralized within the `IAccessControlAsService` contract, offers a resilient and adaptable framework for managing access control in contract services. The access control system is characterized by its incorporation of both RBAC and ABAC principles, which are further enhanced by the dynamic capabilities of state-based access control. This integration yields a strong, context-aware, and highly effective access control mechanism.

7.2 Evaluation

7.2.1 Methodology

1. **Objective:** Evaluate the efficiency of the two smart contract architectures—Factory, and Minimal Proxy—in implementing access control systems. We avoid the single contract implementation as the transaction gas limit on the contract size does not allow putting the whole logic we have for this level of Access Control Management in a single contract. For benchmarking, we also disabled dynamic state-based ABAC policies for both architectures, as the factory equivalent failed to deploy due to the large contract size. Disabling this feature does not affect the metrics comparison for the Factory and Minimal Proxy designs. These results are based on the simple scenario of having one user, one role, one RBAC and one ABAC. In the next sections we evaluate the scalability of our platform based on different metrics.
2. **Benchmarking Tools:** Hardhat and a local Ethereum node for deploying contracts and measuring gas costs.
3. **Test Environment:** Apple M1 Pro, 16 GB RAM, MacOS.
4. **Scenarios:** Two different access control complexity levels are considered:
 - (a) Simple Role-Based Access Control (RBAC).
 - (b) Role Centric(RBAC with Attribute-Based Access Control (ABAC)).

7.2.2 Metrics

1. **Deployment Costs:** Gas units required for deploying the contracts.

2. **Admin Transaction Costs:** Gas units consumed for administrative tasks such as adding/removing roles or attributes.
3. **User Transaction Costs:** Gas units consumed for users to perform actions like accessing resources.

	First		All Next Deployments	
	Factory	Clone	Factory	Clone
Deployment	8,873,299	6,132,183	4,036,095	1,050,801

Table 7.1 Benchmarking results for two architectures. All costs are in gas units.

	Scenario 1		Scenario 2	
	Factory	Clone	Factory	Clone
Admin Transaction	114,993	120,569	105,913	113,067
User Transaction	117,307	125,391	77,507	85,591

Table 7.2 Benchmarking results across two scenarios and two architectures. All costs are in gas units.

7.2.3 Discussion

In comparing the Factory and Clone (Minimal Proxy) architectures for implementing access control systems in smart contracts, we observe different performance trade-offs based on deployment and transaction types.

1. Deployment Costs:

- *Initial Deployment:* The Factory architecture exhibits higher gas consumption at 8,873,299 units, approximately 30% more than the 6,132,183 units required by the Clone architecture. This suggests that the Clone architecture may be preferable from a gas efficiency perspective for projects that only necessitate a single or few deployments.
- *Subsequent Deployments:* For applications that require multiple deployments, the Clone architecture's benefits become more pronounced. It necessitates only about a quarter of the gas that the Factory architecture does (1,050,801 units compared

to 4,036,095 units). Therefore, in scenarios where repeated deployments are anticipated, the Clone architecture emerges as the clear front-runner regarding gas efficiency.

2. Transaction Costs:

- *Admin Transaction Costs:* Across both scenarios, the Factory architecture has a slight edge over the Clone architecture, with gas consumption differences ranging from 1% to 7% in favour of the Factory architecture. These findings suggest that the Factory might be preferable if administrative tasks dominate.
- *User Transaction Costs:* The Clone architecture consumes more gas for both scenarios than the Factory architecture. The difference ranges from 7% to 10%, indicating that the Factory architecture is more efficient for user-based transactions.

Our evaluation of the Factory and Minimal Proxy patterns showed that the deployment gas cost is notably reduced in the Minimal Proxy design. While there is a slight uptick in gas consumption for user and administrative interactions with the Minimal Proxy, this increase is negligible and does not constitute a significant overhead. It's worth highlighting that our benchmarks exclusively focused on the Factory and Minimal Proxy designs, sidelining the Single Contract solution. Though, in theory, the Single Contract solution might offer reduced gas costs for user interactions due to the absence of external calls, encapsulating the intricate logic of role-centric access control within a singular contract proved infeasible. This limitation stemmed from Ethereum's constraints on contract size, making implementing such a comprehensive system impractical without partitioning the logic.

7.3 Scalability of Role-Centric Access Control as a Service Platform

Our platform's scalability was rigorously evaluated through benchmarks focusing on Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) mechanisms. These tests were designed to assess the impact of increasing the number of roles and ABAC policies on gas consumption in a blockchain environment.

7.3.1 Benchmark Setup

The benchmark scenario based on a Storage smart contract similar to the one presented in Listing 7.1 and equipped with our access control service setup as shown in Listing 7.2, deployed alongside the access control service contracts.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.20;
3
4 import "./Clone/NS/IACaaSAccessControlNS.sol";
5
6 contract Storage is IACaaSAccessControlNS{
7     uint256 storedData;
8     string storedString;
9     mapping (address => uint) public balances;
10    constructor(address _acsAddress) {
11        _ACaaS_initialize(_acsAddress);
12    }
13
14    function set(uint256 x,string memory y) public checkAccess{
15        storedData = x;
16        storedString = y;
17    }
18
19    function get() public checkAccess view returns (uint256) {
20        return storedData;
21    }
22
23    function addBalance(uint256 _balance) public{
24        balances[msg.sender] = _balance;
25    }
26 }

```

Listing 7.2 Sample Storage Contract

We perform the benchmarks on the access control operation, which is the most critical function, engaging multiple queries to user roles, attributes, RBAC, and ABAC policies.

We do the benchmark regarding one user and one resource(the set() function). As we discussed, changing the number of users and resources does not impact the gas cost of access control operation.

7.3.2 Results

Gas Cost vs. Number of ABACs with Fixed RBACs

Figure 7.5 illustrates the gas costs associated with increasing ABAC policies while keeping the number of RBAC policies fixed at 10.

- With a fixed number of RBACs (10), there is a clear increase in gas cost as the number of ABACs increases.
- This trend suggests a linear or near-linear scalability with the number of ABACs.
- The system's predictable increase in gas cost with additional ABAC policies is an important factor in design and planning, especially in environments with a potential for numerous policies.

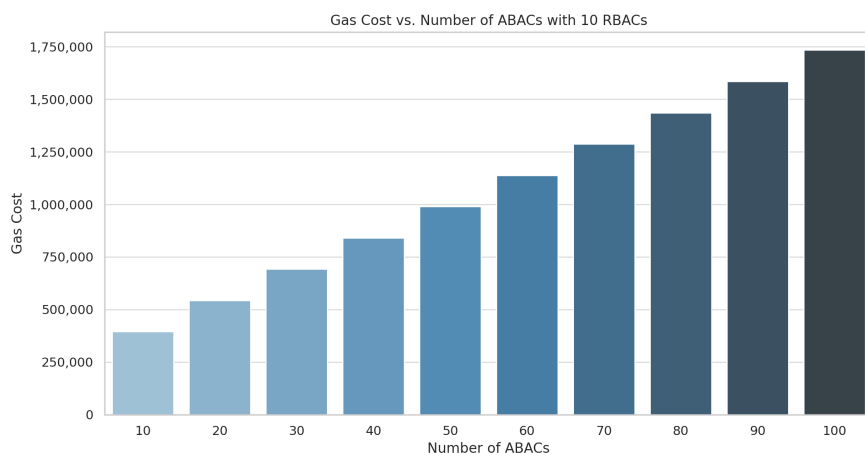


Figure 7.5 Gas Cost vs Number of ABACs

Gas Cost vs. Number of RBACs with Fixed ABACs

Conversely, Figure 7.6 demonstrates the effect of escalating the number of RBAC policies with a constant number of 10 ABAC policies.

- A similar increase in gas cost is observed with the increasing number of RBACs, keeping the number of ABACs fixed (10).

- The relationship appears linear or nearly linear, indicating predictable cost increases with more roles.
- In large-scale systems requiring many roles, this scalability aspect is key, as it implies higher gas costs with additional roles.

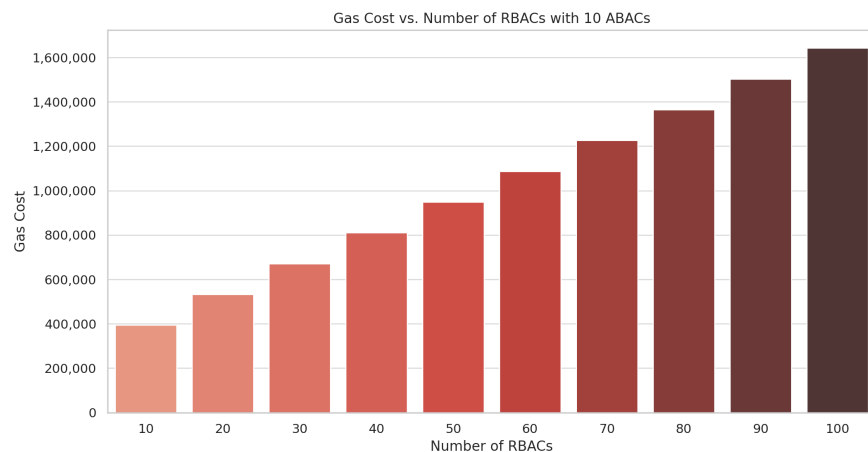


Figure 7.6 Gas Cost vs Number of ABACs

7.3.3 Overall Scalability

- The system shows linear or near-linear scalability with the number of RBACs and ABACs.
- This scalability, while beneficial for predictability, means that large-scale implementations might face significant cost implications.
- Efficient management of roles and policies is necessary for cost control in large-scale deployments.

7.3.4 Considerations for Large-Scale Implementations

- Large-scale deployments require careful planning to balance the number of roles and policies against gas costs.
- Optimizing the system by consolidating roles and policies where possible can enhance efficiency and cost-effectiveness.

7.4 Use Case: Managing Margin in Perpetual Protocol

7.4.1 Introduction: The Perpetual Protocol

Perpetual Protocol is a decentralized derivatives exchange offering perpetual contracts on the Ethereum blockchain. The protocol employs Virtual Automated Market Makers for liquidity and allows traders to take long or short positions without an expiry date. Effective margin management ensures fair and secure trading on this platform.

7.4.2 The Current Issue

Managing margin is a critical aspect of any derivatives trading platform. Incorrect or unauthorized modifications to margin settings could lead to unintended liquidations or system abuse, posing risks to the protocol and its users. Thus, there is a need for a secure, robust, and adaptable access control mechanism.

We propose the implementation of our Role-Centric Access Control as a Service for secure and efficient management of margin settings in Perpetual Protocol.

Integration with ClearingHouse Contract

Here, we provide an example of extending the `ClearingHouse` to make it compatible with our platform.

```

1 contract ClearingHouse is
2     ...,
3     IACaaSAccessControl {
4     ...
5     function initialize(
6         ...
7         address _acsAddress
8     ) public initializer {
9         ...
10        _ACaaS_initialize(_acsAddress);
11        ...
12    }
13
14    function addLiquidity(AddLiquidityParams calldata params){
15        ...

```

```

16         checkAccess
17         ...
18     {
19         ...
20     }
21
22     function removeLiquidity(RemoveLiquidityParams calldata
23     params){
24         ...
25         checkAccess
26         ...
27     {
28         ...
29     }
30     ...
31 }

```

Listing 7.3 ClearingHouse Integration.

As you can see, the changes are minimal, and after contract deployment, anything related to access control will be managed via our access control service provider deployed in `_acsAddress` address.

Let's consider an admin's scenarios for adding and removing liquidity.

An RBAC policy might define the `LIQUIDITY_PROVIDER` role and grant this role to liquidity providers. This is essential as our access control model is role-centric. So, we define RBACs and define ABACs to restrict these roles.

Following the deployment of the modified ClearingHouse contract, administrators can dynamically manage access control policies via the access control service provider located at `_acsAddress`.

Time-Based Restriction Scenario

Suppose the system administrator recognizes the potential of time-based restrictions to enhance the protocol. Here are some advantages that could be gained:

- **Reduced Slippage:** Restricting `addLiquidity` and `removeLiquidity` operations to specific

times can result in more predictable liquidity levels, reducing slippage during high-volume trading periods.

- **Risk Mitigation:** Limiting liquidity operations to predefined time slots minimizes exposure to market manipulation tactics, such as flash loans or front-running.
- **Regulatory Alignment:** Time-based restrictions can help the protocol comply with various trading time rules that regulators may impose, offering a more harmonious operating environment.

The admin would use the `addRBACABACs` function from our service layer to implement these time-based restrictions. This enables the flexibility to dynamically set or modify time-based controls without altering the core `ClearingHouse` smart contract.

```

1 contract TimeBasedPolicy {
2     function isTimeWithinRange(uint256 startTime, uint256
3     endTime) public view returns (bool) {
4         return block.timestamp >= startTime && block.timestamp
5         <= endTime;
6     }
7 }

```

Listing 7.4 Custom Time Based Policy Contract

And add the required policy via a transaction with this specific function call:

```

1 addRBACABACs (
2     RBACId,
3     address(timeBasedPolicyContract),
4     bytes4(keccak256("isWithinAllowedHours()")),
5     abi.encode(),
6     "bool",
7     abi.encode(true),
8     false
9 );

```

Listing 7.5 Add time-based ABAC policy

The `addRBACABACs()` function is a part of our platform and represents a concrete implementation of state-based ABAC access controls. This detailed example shows how you can leverage the granularity and flexibility of our access control service.

7.5 Conclusion and Future work

This research proposes an Access Control as a Service platform for smart contracts' functionality on Ethereum. Ethereum is a secure and auditable blockchain that provides the opportunity to create decentralized applications using smart contracts. The number of smart contracts implemented on Ethereum is growing fast. One of the main concerns about the DApps on Ethereum is the access control of their functions. Access control solutions like the OpenZeppelin access control library deal with some limitations. The tight coupling of the core business logic of decentralized applications and the access control logic, and the low possibility or impossibility of updates or modifications to the access control system after deployment are some of the limitations of the current access control management methods. To address these limitations, we propose Blockchain-Based Access Control as a Service to manage the smart contracts' access control at the function level on Ethereum. Our proposed model decouples the core business logic and the access control logic of DApps and releases the DApp developers from the access control management details. In addition, our platform allows updating and modification of access control policies, roles, and user attributes. Furthermore, this platform maintains the auditability of the access control system by storing the access control policies transparently on the blockchain. In addition, we use a combination of role-based access control and attribute-based access control models to create a role-centric access control model. This model will benefit from the simplicity and ease of management of RBAC while maintaining ABAC's flexibility.

Another distinguishing component of our platform is the state-based dynamic ABAC definition, which enables the definition of ABAC rules based on the current state of the blockchain.

As part of our future work, the correctness of access control contracts can be formally verified. In addition, there are different techniques for protecting users' privacy in an access control management platform, like zero-knowledge proof.

CHAPTER 8 CONCLUSION

This thesis represents a significant stride in blockchain technology and smart contract security. It underscores the integration and enhancement of access control mechanisms within Ethereum smart contracts and the innovation in detecting errors in Halo2 circuit implementations. The broader implications of this work extend beyond the immediate solutions provided, setting a foundation for future advancements in blockchain applications and security.

Automated Analysis of Halo2 Circuits

The second crucial contribution of this thesis is developing a tool and techniques for identifying errors in the Halo2 circuits. This effort is pivotal in ensuring the reliability and security of Ethereum ZK rollups. Our tool adeptly detects under- and over-constrained circuits. These detections are crucial in preempting potential errors in Halo2 circuit implementations. Although this work is in its proof-of-concept stage, it signifies a breakthrough in the automated analysis of Halo2 circuits, an unexplored area fraught with challenges.

halo2 analyzer: static design and Cycle Notion-based Optimization

In this work, we introduced a novel design for the halo2 analyzer, leveraging a custom Assignment trait for static circuit analysis. Our approach employs an Assembly data structure to articulate the circuit's copy constraints through multiple matrices, facilitating the representation of cell cycles. We significantly reduced the SMT variables and constraints using these cycles, markedly improving execution time. This optimization marks a substantial advancement in the efficiency of circuit analysis, setting the stage for further innovations in the field.

Abstract Interpretation for PLONKish Circuits

In this work, we have advanced the field of abstract interpretation applied to PLONKish circuits, focusing on optimizing the analysis of such circuits within Ethereum's cryptographic frameworks. Our innovative methods facilitate the handling of copy constraints and lookup arguments, enhancing the precision of detecting under-utilized elements in circuit design. By implementing an abstract interpretation framework that efficiently processes these constraints, we have significantly decreased the occurrence of false negatives, thereby increasing

the reliability of circuit analyses. This contribution not only improves the accuracy of circuit evaluations but also sets a precedent for future research on expanding the capabilities of abstract interpretation in blockchain technologies. The potential to further refine and automate these techniques could dramatically influence the security and efficiency of Ethereum ZK rollups and similar applications.

Access Control as a Service for Ethereum Smart Contracts

The development of an Access Control as a Service platform tailored for Ethereum smart contracts marks our first major contribution. This novel approach transcends the limitations of existing solutions like the OpenZeppelin access control library by effectively decoupling the core business logic of decentralized applications (DApps) from their access control logic. Our proposed Blockchain-Based Access Control as a Service not only eases the development process for DApp creators but also introduces a new paradigm in managing access control at the function level. This system allows for post-deployment updates and modifications, a significant leap forward in smart contract management. Integrating role-based access control (RBAC) with attribute-based access control (ABAC), enriched by our innovative state-based dynamic ABAC model, greatly enhances both the flexibility and the auditability of access control systems, paving the way for more secure and efficient DApps.

Future Directions

The horizon brims with potential research avenues as we gaze into the future. In the domain of Halo2 circuit analysis, the scalability of our current tooling and the inclusion of more robust analyses to address a wider spectrum of issues are imperative next steps. For the Access Control as a Service platform, venturing into the realms of formal verification of access control contracts and incorporating privacy-preserving mechanisms like zero-knowledge proofs are of paramount interest. Moreover, tackling the inference problem in blockchain-based access control systems presents a fertile ground for future exploration. These upcoming initiatives hold the potential to significantly transform blockchain technology further, enhancing its security and applicability.

Conclusion

This thesis transcends the mere presentation of solutions; it embodies a journey of discovery and innovation in the blockchain landscape. The contributions herein address pivotal

challenges in circuit reliability and smart contract security and lay the groundwork for transformative future innovations. The pioneering work in error detection of Halo2 circuits coupled with the integration of sophisticated access control mechanisms within smart contracts, illustrates the evolving potential of blockchain technology. It is a testament to the promise of blockchain technology to evolve securely and efficiently, meeting the complex demands of our increasingly digital world.

REFERENCES

- [1] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, “Marlin: Pre-processing zkSNARKs with universal and updatable SRS,” in *Advances in Cryptology—EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I* 39. Springer, 2020, pp. 738–768.
- [2] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge,” *IACR Cryptol. ePrint Arch.*, p. 953, 2019. [Online]. Available: <https://eprint.iacr.org/2019/953>
- [3] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct NIZKs without PCPs,” in *Advances in Cryptology—EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26–30, 2013. Proceedings* 32. Springer, 2013, pp. 626–645.
- [4] 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. Springer, 2016, pp. 305–326.
- [5] —, “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*, ser. Lecture Notes in Computer Science, M. Fischlin and J. Coron, Eds., vol. 9666. Springer, 2016, pp. 305–326. [Online]. Available: https://doi.org/10.1007/978-3-662-49896-5_11
- [6] S. Bowe, J. Grigg, and D. Hopwood, “Halo: Recursive proof composition without a trusted setup,” *IACR Cryptol. ePrint Arch.*, p. 1021, 2019. [Online]. Available: <https://eprint.iacr.org/2019/1021>
- [7] A. Pertsev, R. Semenov, and R. Storm, “Tornado cash privacy solution version 1.4,” December 2019. [Online]. Available: <https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf>

- [8] A. Gluchowski, “Introducing zkSync: the missing link to mass adoption of ethereum,” 2021.
- [9] “Zircuit,” www.zircuit.com/. Accessed 2 Feb. 2024. [Online]. Available: <https://www.zircuit.com/>
- [10] “Scroll.” [Online]. Available: <https://scroll.io/>
- [11] Polygon, “Polygon. polygon zkevm,” 2023. [Online]. Available: <https://polygon.technology/polygon-zkevm>
- [12] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct nizks without pcps,” in *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26–30, 2013. Proceedings 32*. Springer, 2013, pp. 626–645.
- [13] P. Wackerow. Layer 2 rollups. [Online]. Available: <https://ethereum.org/en/developers/docs/scaling/layer-2-rollups/>
- [14] ZCash, “Halo2,” 2023, <https://github.com/zcash/halo2>.
- [15] F. Wang, “Ecne: Automated verification of zk circuits,” Feb 2022. [Online]. Available: <https://0xparc.org/blog/ecne>
- [16] Y. Chen, “Picus,” April 2022. [Online]. Available: <https://github.com/Veridise/Picus>
- [17] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. V. Gaffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, “Automated detection of underconstrained circuits for zero-knowledge proofs,” Cryptology ePrint Archive, Paper 2023/512, 2023. [Online]. Available: <https://eprint.iacr.org/2023/512>
- [18] “Tornado.cash got hacked. By us.” October 2019. [Online]. Available: <https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8>
- [19] M. Connor, “Disclosure of recent vulnerabilities,” 2021. [Online]. Available: <https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities>
- [20] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International*

- Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24
- [21] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252. [Online]. Available: <https://doi.org/10.1145/512950.512973>
- [22] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, “Decentralized applications: The blockchain-empowered software system,” *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018.
- [23] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, “Blockchain-enabled smart contracts: architecture, applications, and future trends,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.
- [24] N. Szabo, “Smart contracts : Building blocks for digital markets,” 2018.
- [25] NCCGroup *et al.*, “Decentralized application security project (dasp) top 10,” 2018.
- [26] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.
- [27] Parity *et al.*, “The multi-sig hack: A postmortem,” 2017.
- [28] SlowMist *et al.*, “The root cause of poly network being hacked,” 2021.
- [29] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal, “A comparative survey of symmetric and asymmetric key cryptography,” in *2014 international conference on electronics, communication and computational engineering (ICECCE)*. IEEE, 2014, pp. 83–93.
- [30] A. Salomaa, “Public-key cryptography,” 2013.
- [31] F. J. Aufa, A. Affandi *et al.*, “Security system analysis in combination method: Rsa encryption and digital signature algorithm,” in *2018 4th International Conference on Science and Technology (ICST)*. IEEE, 2018, pp. 1–5.

- [32] P. Rogaway and T. Shrimpton, “Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance,” in *International workshop on fast software encryption*. Springer, 2004, pp. 371–388.
- [33] D. R. Stinson, “Some observations on the theory of cryptographic hash functions,” *Designs, Codes and Cryptography*, vol. 38, no. 2, pp. 259–277, 2006.
- [34] R. C. Merkle, “One way hash functions and des,” in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 428–446.
- [35] M. Bellare and P. Rogaway, “Collision-resistant hashing: Towards making uowhfs practical,” in *Annual International Cryptology Conference*. Springer, 1997, pp. 470–484.
- [36] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Review*, p. 21260, 2008.
- [37] R. J. Maly, J. Mischke, P. Kurtansky, and B. Stiller, “Comparison of centralized (client-server) and decentralized (peer-to-peer) networking,” *Semester thesis, ETH Zurich, Zurich, Switzerland*, pp. 1–12, 2003.
- [38] C. Antal, T. Cioara, I. Anghel, M. Antal, and I. Salomie, “Distributed ledger technology review and decentralized applications development guidelines,” *Future Internet*, vol. 13, no. 3, p. 62, 2021.
- [39] G. Becker, “Merkle signature schemes, merkle trees and their cryptanalysis,” *Ruhr-University Bochum, Tech. Rep*, 2008.
- [40] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun, “A review on consensus algorithm of blockchain,” in *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2017, pp. 2567–2572.
- [41] Ethereum whitepaper. [Online]. Available: <https://ethereum.org/en/whitepaper/>
- [42] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O’reilly Media, 2018.
- [43] C.-P. Schnorr, “Efficient signature generation by smart cards,” *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.
- [44] T. P. Pedersen, “Non-interactive and information-theoretic secure verifiable secret sharing,” in *Annual international cryptology conference*. Springer, 1991, pp. 129–140.

- [45] 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*. Springer, 1986, pp. 186–194.
- [46] S. Bowe, J. Grigg, and D. Hopwood, “Recursive proof composition without a trusted setup,” *Cryptology ePrint Archive*, 2019.
- [47] J. Kilian, “A note on efficient zero-knowledge proofs and arguments,” in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, 1992, pp. 723–732.
- [48] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.
- [49] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, “Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2111–2128.
- [50] P. Wackerow. Scaling. [Online]. Available: <https://ethereum.org/en/developers/docs/scaling/>
- [51] M. Bez, G. Fornari, and T. Vardanega, “The scalability challenge of ethereum: An initial quantitative analysis,” in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 167–176.
- [52] R. Ramesh, “Algorithm for cross-shard cross-EE atomic user-level ETH transfer in ethereum 2,” *arXiv preprint arXiv:2102.09688*, 2021.
- [53] T. Schaffner, “Scaling public blockchains,” 2021.
- [54] K. Floersch, “Ethereum smart contracts in L2: Optimistic rollup,” 2019.
- [55] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-based access control*. Artech house, 2003.
- [56] D. R. Kuhn, E. J. Coyne, T. R. Weil *et al.*, “Adding attributes to role-based access control,” *Computer*, vol. 43, no. 6, pp. 79–81, 2010.
- [57] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone *et al.*, “Guide to attribute based access control (ABAC) definition and considerations (draft),” *NIST special publication*, vol. 800, no. 162, pp. 1–54, 2013.

- [58] X. Jin, R. Sandhu, and R. Krishnan, “Rabac: role-centric attribute-based access control,” in *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. Springer, 2012, pp. 84–96.
- [59] A. Ozdemir, S. Pailoor, A. Bassa, K. Ferles, C. Barrett, and I. Dillig, “Split gröbner bases for satisfiability modulo finite fields,” *Cryptology ePrint Archive*, 2024.
- [60] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, “Sok: What don’t we know? understanding security vulnerabilities in snarks,” *arXiv preprint arXiv:2402.15293*, 2024.
- [61] L. C. Vella and L. Alt, “On satisfiability of polynomial equations over large prime fields,” in *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, ser. CEUR Workshop Proceedings, D. Déharbe and A. E. J. Hyvärinen, Eds., vol. 3185. CEUR-WS.org, 2022, pp. 114–127. [Online]. Available: <http://ceur-ws.org/Vol-3185/extended9913.pdf>
- [62] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng, “Practical security analysis of zero-knowledge proof circuits.” *IACR Cryptol. ePrint Arch.*, vol. 2023, p. 190, 2023.
- [63] Y. Fan, Y. Xu, and C. Garman, “Snarkprobe: An automated security analysis framework for zkSNARK implementations,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2024, pp. 340–372.
- [64] M. Thomas, “Orbis specification language: a type theory for zk-SNARK programming,” *IACR Cryptol. ePrint Arch.*, p. 1003, 2022. [Online]. Available: <https://eprint.iacr.org/2022/1003>
- [65] noir contributors, “noir zksnark language,” 2022. [Online]. Available: <https://aztec.network/aztec-nr>
- [66] zksecurity, “Noname: a programming language to write zkapps,” 2023. [Online]. Available: <https://github.com/zksecurity/noname>
- [67] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith, “Leo: A programming language for formally verified, zero-knowledge applications,” *IACR Cryptol. ePrint Arch.*, p. 651, 2021. [Online]. Available: <https://eprint.iacr.org/2021/651>

- [68] 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,” *Cryptology ePrint Archive*, 2023.
- [69] (2021) Access control. [Online]. Available: <https://docs.openzeppelin.com/contracts/4.x/access-control>
- [70] (2017) Access control. [Online]. Available: <https://dappsys.readthedocs.io/en/latest/index.html#dappsys>
- [71] A. Chatterjee, Y. Pitroda, and M. Parmar, “Dynamic role-based access control for decentralized applications,” in *International Conference on Blockchain*. Springer, 2020, pp. 185–197.
- [72] D. D. F. Maesa, P. Mori, and L. Ricci, “A blockchain based approach for the definition of auditable access control systems,” *Computers & Security*, vol. 84, pp. 93–119, 2019.
- [73] K. Jannes, V. Reniers, W. Lenaerts, B. Lagaisse, and W. Joosen, “Dedacs: Decentralized and dynamic access control for smart contracts in a policy-based manner,” in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 2023, pp. 222–230.
- [74] R. Kumar and R. Tripathi, “Scalable and secure access control policy for healthcare system using blockchain and enhanced bell-lapadula model,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 2, pp. 2321–2338, 2021.
- [75] S. Noh, S. U. Shin, and K.-H. Rhee, “Pyros: A state channel-based access control system for a public blockchain network,” *Security and Communication Networks*, vol. 2020, 2020.
- [76] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, “Fairaccess: a new blockchain-based access control framework for the internet of things,” *Security and communication networks*, vol. 9, no. 18, pp. 5943–5964, 2016.
- [77] M. U. Rahman, B. Guidi, and F. Baiardi, “Blockchain-based access control management for decentralized online social networks,” *Journal of Parallel and Distributed Computing*, vol. 144, pp. 41–54, 2020.
- [78] F. Sabrina, “A novel entitlement-based blockchain-enabled security architecture for iot,” in *2019 29th International Telecommunication Networks and Applications Conference (ITNAC)*. IEEE, 2019, pp. 1–7.

- [79] N. Sivaselvan, V. Bhat, and M. Rajarajan, "Blockchain-based scheme for authentication and capability-based access control in iot environment," in *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. IEEE, 2020, pp. 0323–0330.
- [80] M. A. Bouras, B. Xia, A. O. Abuassba, H. Ning, and Q. Lu, "Iot-ccac: a blockchain-based consortium capability access control approach for iot," *PeerJ Computer Science*, vol. 7, p. e455, 2021.
- [81] M. U. Rahman, B. Guidi, F. Baiardi, L. Ricci, L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa, "Context-aware and dynamic role-based access control using blockchain." in *AINA*, 2020, pp. 1449–1460.
- [82] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the internet of things," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1594–1605, 2018.
- [83] A. Ouaddah and B. Bellaj, "Fairaccess2. 0: a smart contract-based authorisation framework for enabling granular access control in iot," *International Journal of Information and Computer Security*, vol. 15, no. 1, pp. 18–48, 2021.
- [84] A. Ouaddah, "A blockchain based access control framework for the security and privacy of iot with strong anonymity unlinkability and intractability guarantees," in *Advances in Computers*. Elsevier, 2019, vol. 115, pp. 211–258.
- [85] T. Sultana, A. Almogren, M. Akbar, M. Zuair, I. Ullah, and N. Javaid, "Data sharing system integrating access control mechanism using blockchain-based smart contracts for iot devices," *Applied Sciences*, vol. 10, no. 2, p. 488, 2020.
- [86] Y. Zhang, M. Yutaka, M. Sasabe, and S. Kasahara, "Attribute-based access control for smart cities: A smart-contract-driven framework," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6372–6384, 2020.
- [87] N. Shi, L. Tan, C. Yang, C. He, J. Xu, Y. Lu, and H. Xu, "Bacs: A blockchain-based access control scheme in distributed internet of things," *Peer-to-peer networking and applications*, pp. 1–15, 2020.
- [88] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: <https://doi.org/10.1137/0218012>

- [89] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” 2014. [Online]. Available: <https://ethereum.github.io/yellowpaper/paper.pdf>
- [90] P. McCorry, C. Buckland, B. Yee, and D. Song, “SoK: Validating bridges as a scaling solution for blockchains,” *IACR Cryptol. ePrint Arch.*, p. 1589, 2021. [Online]. Available: <https://eprint.iacr.org/2021/1589>
- [91] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, “Commit-chains: Secure, scalable off-chain payments,” *IACR Cryptol. ePrint Arch.*, p. 642, 2018. [Online]. Available: <https://eprint.iacr.org/2018/642>
- [92] M. Bellés-Muñoz, J. B. Melé, V. Daza, and J. L. Muñoz-Tapia, “New privacy practices for blockchain software,” *IEEE Softw.*, vol. 39, no. 3, pp. 43–49, 2022. [Online]. Available: <https://doi.org/10.1109/MS.2021.3086718>
- [93] J. Eberhardt and S. Tai, “ZoKrates - scalable privacy-preserving off-chain computations,” in *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), iThings/GreenCom/CPSCoM/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*. IEEE, 2018, pp. 1084–1091. [Online]. Available: https://doi.org/10.1109/Cybermatics_2018.2018.00199
- [94] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, “HyperPlonk: Plonk with linear-time prover and high-degree custom gates,” *IACR Cryptol. ePrint Arch.*, p. 1355, 2022. [Online]. Available: <https://eprint.iacr.org/2022/1355>
- [95] A. Gabizon and Z. J. Williamson, “The turbo-PLONK program syntax for specifying snark programs,” 2019. [Online]. Available: https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf
- [96] Zac, “Aztec’s ZK-ZK-Rollup, looking behind the cryptocurtain,” March 2021. [Online]. Available: <https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619>
- [97] A. Gabizon and Z. J. Williamson, “plookup: A simplified polynomial protocol for lookup tables,” *IACR Cryptol. ePrint Arch.*, p. 315, 2020. [Online]. Available: <https://eprint.iacr.org/2020/315>

- [98] A. Ozdemir, G. Kremer, C. Tinelli, and C. W. Barrett, “Satisfiability modulo finite fields,” *IACR Cryptol. ePrint Arch.*, p. 91, 2023. [Online]. Available: <https://eprint.iacr.org/2023/091>
- [99] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
- [100] H. M. Ardevol, M. Guzmán-Albiol, J. B. Melé, and J. L. Muñoz-Tapia, “eSTARK: Extending starks with arguments,” *IACR Cryptol. ePrint Arch.*, p. 474, 2023. [Online]. Available: <https://eprint.iacr.org/2023/474>
- [101] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, “SoK: Layer-two blockchain protocols,” in *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Bonneau and N. Heninger, Eds., vol. 12059. Springer, 2020, pp. 201–226. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_12
- [102] N. Szabo, “Smart contracts: building blocks for digital markets,” *EXTROPY: The Journal of Transhumanist Thought*, (16), vol. 18, no. 2, p. 28, 1996.
- [103] “Leo code translates to invalid aleo instruction code,” August 2022. [Online]. Available: <https://github.com/AleoHQ/leo/issues/2042>
- [104] “Proof verification fails with a simple example,” September 2022. [Online]. Available: <https://github.com/noir-lang/noir/issues/358>
- [105] F. H. Soureshjani, M. Hall-Andersen, M. Jahanara, J. Kam, J. Gorzny, and M. Ahmadvand, “Automated analysis of Halo2 circuits,” in *Proceedings of the 21st International Workshop on Satisfiability Modulo Theories (SMT 2023) co-located with the 29th International Conference on Automated Deduction (CADE 2023), Rome, Italy, July, 5-6, 2023*, ser. CEUR Workshop Proceedings, S. Graham-Lengrand and M. Preiner, Eds., vol. 3429. CEUR-WS.org, 2023, pp. 3–17. [Online]. Available: <https://ceur-ws.org/Vol-3429/paper3.pdf>
- [106] “Standard curve database - bn254,” <https://neuromancer.sk/std/bn/bn254>. Accessed 19 Jun. 2024. [Online]. Available: <https://neuromancer.sk/std/bn/bn254>
- [107] D. D. F. Maesa, A. Marino, and L. Ricci, “Detecting artificial behaviours in the bitcoin users graph,” *Online Social Networks and Media*, vol. 3, pp. 63–74, 2017.

- [108] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, “Zerocash: Decentralized anonymous payments from bitcoin,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.
- [109] F. Paci and N. Zannone, “Preventing information inference in access control,” in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, 2015, pp. 87–97.
- [110] (2018) Eip-1167: Minimal proxy contract. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1167>